HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

THESIS

SUBMITTED FOR PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

ENGINEER

IN

INFORMATION TECHNOLOGY

PROTECTED ELASTIC-TREE TOPOLOGIES FOR DATA CENTERS

Author: **Elena Ouro Paz**
Class ICT-60
Supervisor: **Dr. Truong Dieu Linh**

HANOI 05-2016

REQUIREMENTS FOR THE THESIS

1. Student information

Student name: Elena Ouro Paz
Tel: +841297963653 Email: elenaouro@gmail.com
Class: ICT-60 Program: ICT
This thesis is performed at:
From 20/01/2016 to 27/05/2016

2. Goal of the thesis

the main goal of this thesis is the study of the energetic benefits of using protected elastic-tree topologies in data centers while fixing the availability problem that elastic-tree presents. This thesis seeks a possible solution for real data-centers to decrease their energy consumption and still maintain a survivable network in single or double failure scenarios. Through this study we wish to asses the viability of protected elastic-tree topologies and to determine if they present a significant decrease in energy consumption compared to a traditional fat-tree data center that will justify their implementation in real life.
Aside from the main goals of this research, while conducting this thesis, a deeper knowledge of data centers and its architecture will be obtained, together with an understanding of the energetic impact that data centers, in general, and their networks, in particular, entail.

3. Main tasks

Several tasks will be performed to achieve the thesis goals. Three versions of an algorithm will be developed. starting off TAH algorithm (topology aware heuristics) as conceived in the paper *"ElasticTree: Saving Energy in Data Center Networks"* this thesis will comprise the implementation in Scilab and Java of the original algorithm and two variations that will protect the network against one and two failures respectively. Assuming that we know the traffic demand for the data center beforehand the output obtained from the algorithms will contain the information about which links and switches should be active to accommodate all the traffic and to protect from failure. Additionally the output will also contain a series of statistics, including the number of switches active, the number of ports, the power consumed and the utility of the network. From the statistics a through study of the energy consumption by the three versions of the algorithm can be conducted together with a comparison between this results and the power used in a data center with fat-tree topology for the same traffic.
The research will conclude assessing the viability of the protected elastic-tree topology in real data-centers from the results obtained. Considering if the improvement on availability justifies the decrease of energy saved in an unprotected elastic-tree and if regardless of this decrease protected elastic-tree is still more efficient that traditional fat-tree data centers.

4. Declaration of student:

I -Elena Ouro Paz - hereby warrants that the Work and Presentation in this thesis are performed by myself under the supervision of Dr. Truong Dieu Linh.

All results presented in this thesis are truthful and are not copied from any other work.

Hanoi, 27/05/2016

Elena Ouro Paz

5. Attestation of the supervisor on the fulfillment of the requirements of the thesis:

Hanoi, 27/05/2016

Dr. Truong Dieu Linh

ABSTRACT OF THESIS

Traditional data centers represent a significant portion of the yearly power consumption all around the world, many solutions have been submitted to solve this issue such as the elastic-tree topology, a dynamic network structure that suggests that we deactivate unnecessary links and switches in a fat-tree data center, but with such solutions the availability in the network suffers severely. This paper studies the possibility of improving the elastic-tree topology adding single and double failure path protection to make the network more survivable while still saving energy. To achieve this, three algorithms that will find the minimum active topology necessary for a given traffic matrix will be implemented, one without path protection, one for any single failure and the last one for double failure path protection. The results obtained from all three algorithms show that the protected elastic-tree topology that we propose makes the data centers survivable while still saving a significant amount of energy.

TABLE OF CONTENTS

TABLE OF FIGURES

Figure 17	Scenario with two failures between the aggregation and edge layers affecting both working and backup path for k=6.
Figure 18	Power saving levels with middle traffic k=4
Figure 19	Power saving levels with middle traffic k=6
Figure 20	Power saving levels with far traffic k=4
Figure 21	Power saving levels with far traffic k=6
Figure 22	Power saving levels with mixed traffic k=4
Figure 23	Power saving levels with mixed traffic k=6

TABLE OF TABLES

INTRODUCTION

MOTIVATION

Electricity has been embedded in our daily life for years, many of the devices that we use every day, and now deem indispensable, are powered by it and as a result the electricity consumption is increasing throughout the years. However, we must remember that the sources from where it is mostly obtained are not endless and that they often bring devastating consequences to our environment. Governments, organizations and individuals across the world urge us to do what is in our power to decrease our energy consumption. A considerable amount of electricity is consumed in data centers, facilities that provide the necessary infrastructure to process information and offer internet services. For this purpose data centers remain turned on twenty four hours a day powered entirely by electricity. In the USA only, data centers consumed 91 billion kilowatt-hour in 2013 and it is expected to increase up to 140 billion by 2020 [1]. This alarming consumption has moved the scientific community to try and create more efficient data centers. Most efforts have focused on servers and cooling, this amounts to a total of about 70% of the electricity consumption in a data center.

This thesis focuses on another important part of a data center that amounts to about 10-20% of the electricity used, the network.

Solutions using elastic-tree topology[5] have been proposed for fat-tree[4] data centers, these solutions deactivate unnecessary links but they ruin the availability characteristic of fat-tree topologies making the network vulnerable to failures.

This thesis proposes a change in the structure of the network, using a protected elastic-tree topology whose links and switches can be dynamically turned on and off to support all traffic in the data center without consuming unnecessary electricity

ensuring that the data center's network is survivable. This means that in case of a component's failure all traffic demands are still satisfied, fixing the availability problem of elastic-tree topologies. This thesis will also study the viability of such topology and present an evaluation of the energy consumed by a data center with protected elastic-tree with protection for one failure, two and none compared to a traditional data center with a fat-tree topology.

GOALS

the main goal of this thesis is the study of the energetic benefits of using protected elastic-tree topologies in data centers while fixing the availability problem that elastic-tree presents. to achieve this goal three versions of an algorithm will be developed. starting off TAH algorithm (topology aware heuristics) as conceived in the paper *"ElasticTree: Saving*

Energy in Data Center Networks" this thesis will comprise the implementation in Scilab and Java of the original algorithm and two variations that will protect the network against one and two failures respectively. Assuming that we know the traffic demand for the data center beforehand the output obtained from the algorithms will contain the information about which links and switches should be active to accommodate all the traffic and to protect from failure. Additionally the output will also contain a series of statistics, including the number of switches active, the number of ports, the power consumed and the utility of the network. From the statistics a through study of the energy consumption by the three versions of the algorithm can be conducted together with a comparison between this results and the power used in a data center with fat-tree topology for the same traffic.
Aside from the main goals of this research, while conducting this thesis, a deeper understanding of data centers and its architecture will be obtained, together with an understanding of the energetic impact that data centers in general, and their networks in particular entail.

THESIS ORGANIZATION

This thesis consists of four chapters, project planning, problem statement, approach and proposed solution. Additionally it also includes an introduction, conclusions, references and an appendix.
Below there is a brief description of the contents of each section present in this paper:

- Introduction: the introduction is the section you are currently reading. It familiarizes the reader with the topic of the thesis and the structure of the paper. It also briefly describes what has motivated its performance and what it seeks to achieve.

- Project planning: contains the work division that has been made in order to complete the thesis in a suitable time. It contains two gantt charts one indicating the estimated time of each task and one indicating the amount of time that has finally been dedicated to each task at the end of the thesis.

- Problem statement: this chapter of the thesis presents in more detail the problem to be solved during the course of the thesis. This thesis studies the possibility of protecting an elastic-tree topology against single or double failures and therefore has two problems. For each one this chapter will present the input parameters necessary to solve the problem, the output obtained and constraints that the solution must satisfy.

- Approach: the approach chapter gives a background presenting necessary previous knowledge to develop the thesis. It contains information regarding previously existing technologies and related research.

- Proposed solution: Is the most important chapter in this paper. Contains all information regarding the solution to the problem presented in the chapter: “problem statement”. It gives a thorough description of the algorithms that have been developed for single and double failure protection and their implementation. Additionally includes the implementation and design of the fat-tree structure used in both the algorithms mentioned above. In this sec

- Conclusions: what has been concluded from the research after analyzing the results obtained.

- References: sources used in the performance of this thesis.

- Annex: contains the explained code of the algorithms developed in this thesis.

CHAPTER 1: PROJECT PLANNING

The work to be done during this thesis has been divided in 4 work packages with defined tasks for each of them. In the following section the details on work packages and their tasks, as well as a gantt with the estimation of the time each one will take, are presented.

WP1: Algorithms

The work package number one includes all efforts needed to successfully obtain all the versions of the algorithm to later evaluate the results obtained from them.

Tasks:

- TAH in pseudocode
- single failure protected TAH in pseudocode
- double failure protected TAH in pseudocode
- fat-tree structure implementation
- TAH implementation
- single failure protected TAH implementation
- double failure protected TAH implementation
- TAH testing
- single failure protected TAH testing
- double failure protected TAH testing
- troubleshooting

WP2: Evaluation

The second work package comprises all the evaluations done from the results obtained from the above mentioned algorithms. The tasks are divided in 2 phases, corresponding to two different evaluations, the first one (1) containing only the results for TAH and single failure protected TAH. The second phase (2) evaluation contains the results from the three algorithms.

Tasks:

- Obtain and process results (1)
- Energy evaluation (1)
- Obtaining and process results(2)
- Energy evaluation (2)
- Resources evaluation

WP3: Documentation

The third work package contains, as the name indicates, all tasks related to the documentation of the thesis.

Tasks:

- Writing of the thesis

- Preparation of visual aids for the defense of the thesis

WP4: Research

The work package four, is and additional package. Research is conducted throughout the whole period of time when this thesis is carried out but the tasks included in this work package show the periods of time when research is the only activity being performed.

Tasks:

- Initial research
- Research of the necessary changes to protect TAH in case of double failure

2.1 INITIAL SCHEDULE

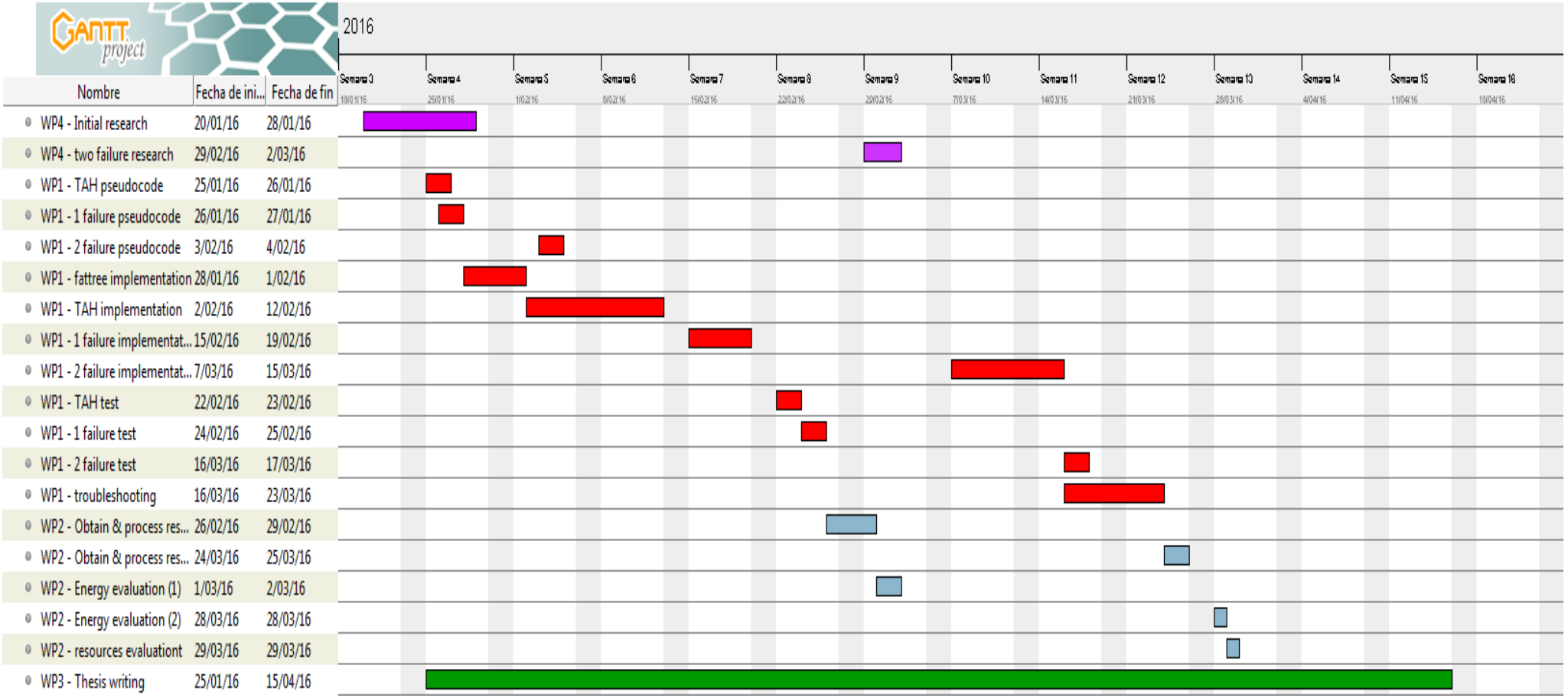

Figure 1: Initial gantt chart

2.2 FINAL SCHEDULE

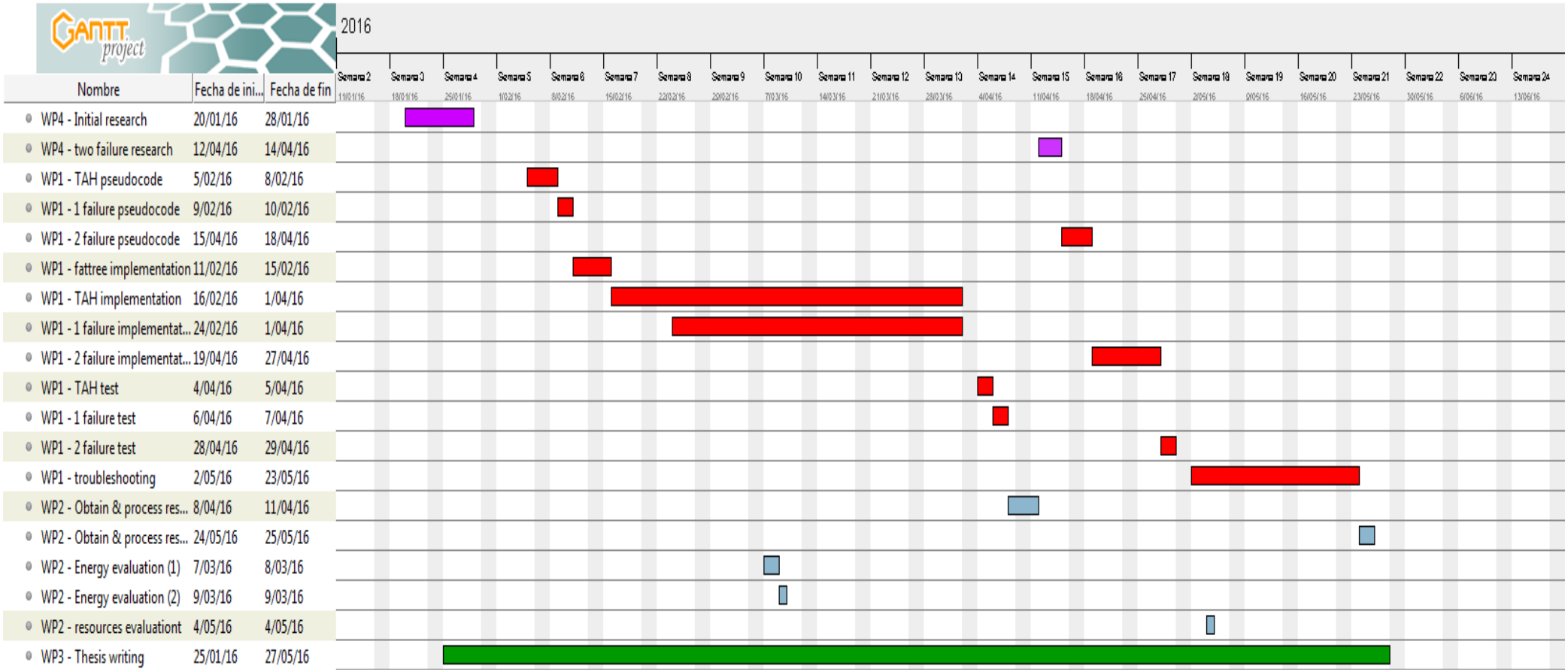

Figure 2: Final gantt chart

CHAPTER 2: PROBLEM STATEMENT

2.1 SINGLE FAILURE PROTECTION

The first problem this thesis tackles is to equip the elastic-tree topology with single failure path protection. Understanding single failure protection as the existence of a backup path that will ensure communications in case that at any given time only one link in the network above the edge layer suffers a failure. Several failures can succeed each other but always having the previous one been fixed beforehand. An algorithm, based in TAH, is developed to find the minimum active topology in a fat tree network that satisfies a given traffic load. Given a fat-tree data center and a traffic matrix containing all traffic requests we output the elements that conform the minimum active topology that will satisfy the given traffic matrix. The following section holds all the information regarding the single failure problem.

2.1.1 ASSUMPTIONS

Some assumptions have been made for simplicity:

- We assume all switches and links are identical
- Switches have k ports
- link rate is r
- We consider the traffic to be perfectly divisible
- We assume the network is a homogeneous fat tree with one link between each pair of connected switches
- A traffic matrix between edge switches is known in advance
- There is at most one failure in the data center at any one time.
- All servers are active.

2.1.2 INPUT

We are given a three-layered homogeneous fat-tree data center where initially only the elements pertaining to the minimum spanning tree are active and a traffic matrix containing all the traffic requests between servers of the data center and their requested bandwidth.

2.1.3 OUTPUT

We search for the minimum active topology that will satisfy the traffic matrix. We obtain a list of all elements that are part of this minimum active topology including switches in the

core, aggregation and edge layers and the links interconnecting them. We additionally obtain statistics to later be analyzed. The statistics will contain the number of active switches, the number of active ports calculated as twice the active links, the power consumption calculated as explained in section 3.5 of this paper and the utilisation index that represents the load in the data center.

2.1.4 CONSTRAINTS

There are several conditions that should be met to ensure the algorithm works correctly:

- We should have $k \geq 4$to have a homogeneous three-layered fat-tree.
- The algorithm will consider a minimum of 20 switches and 16 hosts in the case of k=4 and 48 links interconnecting them.
- Each link will have a minimum of 1 Gbps.
- The bandwidth of any traffic request shall not be greater than the link's capacity.

2.2 DOUBLE FAILURE PROTECTION

The second problem that we propose in this thesis is the improvement of the elastic tree to protect it against double failure. We understand double failure protection of the existence of sufficient backup paths to ensure communications in the case that at any given time two links in the data center suffer from failure. Our algorithm will ensure protection for any two failures that are situated in different pods and two different failures in core-edge links. We do not protect from failures connecting an edge switch and a server in any case.
Given a homogeneous three-layered fat-tree data center and a traffic matrix containing all traffic requests the algorithm outputs the elements that conform the minimum active topology needed to satisfy the traffic matrix. This section contains all information regarding the double failure problem:

2.2.1 ASSUMPTIONS

As in the single failure algorithm we have made some assumptions for simplicity:

- We assume all switches and links are identical
- Switches have k ports
- link rate is r
- Flows are perfectly divisible
- We consider the traffic to be perfectly divisible
- We assume the network is a homogeneous fat tree with one link between each pair of connected switches
- A traffic matrix between edge switches is known in advance

- There is at most two failures in the data center at any one time. Two failures in different pods or two failures in core-edge switches.
- We assume the degree of the fat tree k≥ 6.
- All servers are active.

2.2.2 INPUT

We maintain the same input used in the single failure problem.

2.2.3 OUTPUT

The output remains the same as in the single failure problem including all the active elements in the minimum active topology needed to satisfy the traffic matrix and protect from a double failure as well as the statistics obtained from those results.

2.2.4 CONSTRAINTS

There are several conditions that should be met for the algorithm to work correctly:

- We should have $k \geq 6$to have a homogeneous three-layered fat-tree.
- The algorithm will consider a minimum of 45 switches and 54 hosts in the case of k=6 and 162 links interconnecting them.
- Each link will have a minimum of 1 Gbps.
- The bandwidth of any traffic request shall not be greater than the link's capacity.

CHAPTER 3: APPROACH

3.1 STATE OF THE ART

Nowadays many network solutions have been proposed for the energy problem in data centers, including VL2[6], PortLand[7], DCell[8], and BCube[9]
This thesis will explores the possibility of improving a particular solution, the elastic-tree topology, that will be described in detail in section 3.3. The elastic-tree topology is a solution proposed for fat-tree data center, this topology will be introduced in section 3.1. Fat-tree is a three level tree topology formed by the core, aggregation and edge layers. The switching and routing devices from this three layers (switches from now on) are richly interconnected supporting any communication pattern and it is characterized by a high availability. The downside of using fat-tree topologies is that having such a richly connected network many of the links and switches become redundant and unnecessary for a certain amount of traffic being only an energetic expense. Elastic-tree developers, seeking to improve this energetic issue, propose to disable those unnecessary links and switches in a way that all traffic demands are accommodated but no electricity is wasted.

3. 2 DATA CENTERS

"A data center is a facility that centralizes an organization's IT operations and equipment, and where it stores, manages, and disseminates its data. Data centers house a network's most critical systems and are vital to the continuity of daily operations." [12] Is the definition that paloalto networks gives for the term data centers, is a very simple definition but it gives us a first peek into the diversity of data centers.
Data centers [11] have very diverse roles, they can be used as campus networks, private WAN, remote access, Internet server farm, Extranet server farm or Intranet server farm. Since they can serve different purposes they have to be designed accordingly. There are several traits that should be considered when designing a data center:

- Availability: ensuring that data continues to be available at a required level of performance regardless of the situation.[13]
- Scalability: refers to the extent to which some system, component or process is scalable. This means that the work that can be performed changes roughly in proportion to the change in size of the input[14]
- Security: is the protection of information assets.[15]
- Performance: is the amount of work accomplished by a computer system. [16]

- Manageability: measure of and set of features that support the ease, speed, and competence with which a system can be discovered, configured, modified, deployed, controlled, and supervised.[17]

3.3 FAT-TREE TOPOLOGY

The fat tree[18] network is a class of universal routing invented by Charles E. Leiserson in 1985.
The fat tree is a topology based on a complete binary tree. It interconnects several processors through a tree structure. The processors are located at the leaves of the tree while the internal nodes are switches in charge of routing the traffic between processors. The connections between nodes in the tree increase the closer we get to the core node.
The fat-tree used in this thesis differs from the original. A k-ary fat tree has been adopted. Figure 3 shows the organization of a k-ary fat tree for a value of k=4.
A k-ary tree is organized in three levels, the core, the aggregation and the edge layer. Each of the layers has a certain amount of switches in charge of routing the information between servers.
In a k-ary tree we have:

- $(k/2)^2$ Core nodes
- k PODs containing:
  - $k/2$ aggregation switches
  - $k/2$ edge switches
- $k^3/4$ servers

Each core node is connected to one aggregation switch in every POD. Since we have $(k/2)^2$ core nodes and only $k/2$ aggregation switches per POD we cluster the node cores in groups of $k/2$ nodes, inside a group all nodes have the same connections. This redundant connections protect the data center against failure ensuring a high availability.
Aggregation and edge switches inside a POD are completely interconnected.
Each edge switch is connected to $k/2$ servers that are the leaf nodes of the fat tree.

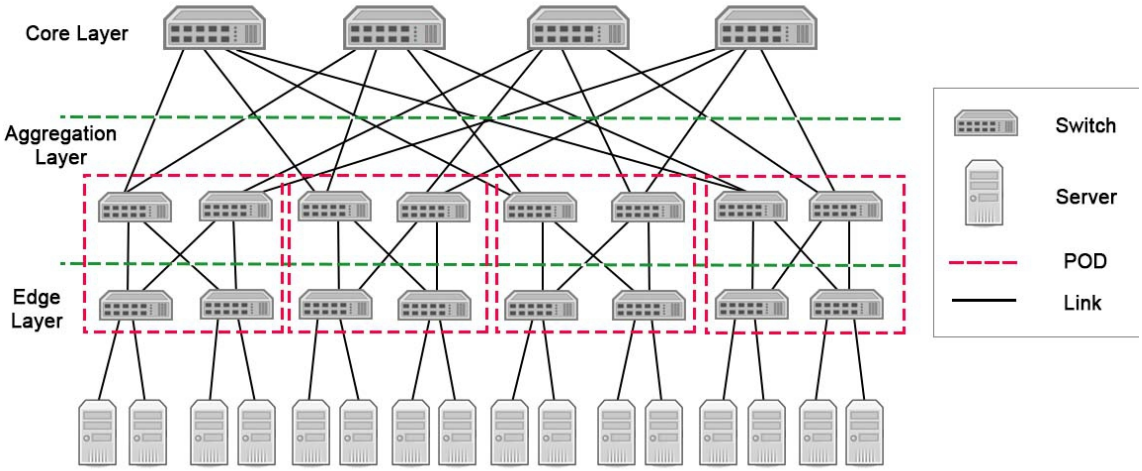

Figure 3: Example of a fat-tree for k=4

3.4 ELASTIC TREE TOPOLOGY

The Elastic Tree, as defined in the original paper, *"ElasticTree: Saving Energy in Data Center Networks",* is a system for dynamically adapting the energy consumption of a Fat-tree data center network.

Elastic tree works on a fat-tree data center, it disables the links and switches that are not being used to route traffic resulting in a network that is a subset tree of the original .Figure 4 shows the minimal spanning tree when using elastic tree, this means that it shows the elements that are active in any case to interconnect all servers in the data center. The rest of links and switches will remain active only if they are needed to route the traffic.

Elastic tree consists of three logical modules as shown in figure 5:

- Optimiser:_is in charge of finding the minimum power network subset to satisfy the given traffic. This means it finds the subset tree of the fat tree that has enough capacity to route all traffic and minimal energy consumption.

  The optimiser takes as input the fat tree, a traffic matrix indicating the source and destination servers and the size of the data that needs to be transmitted between them. It outputs the set of the active components that form the subset tree.

  The Optimiser outputs the result to both the routing and the power control.

  The method used in this thesis to compute a minimum-power network subset is a simple heuristic approach, the topology-aware heuristic that is explained in detail in section 4.1.

- Routing: finds paths for all the traffic flows and pushes routes into the network.
- Power control: power control toggles the power states of ports, linecards and switches.

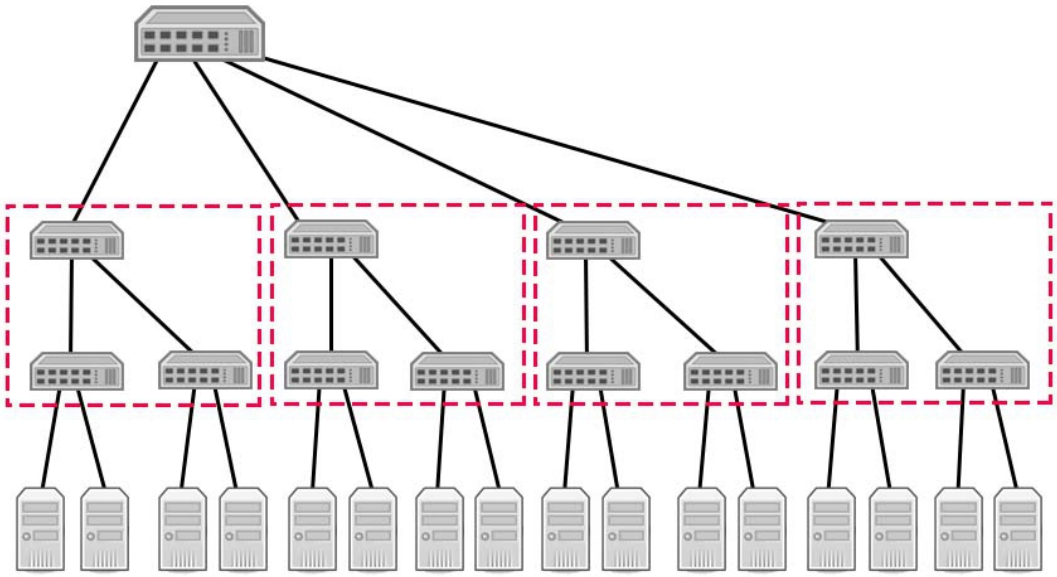

Figure 4: Minimum spanning tree obtained from a fat-tree

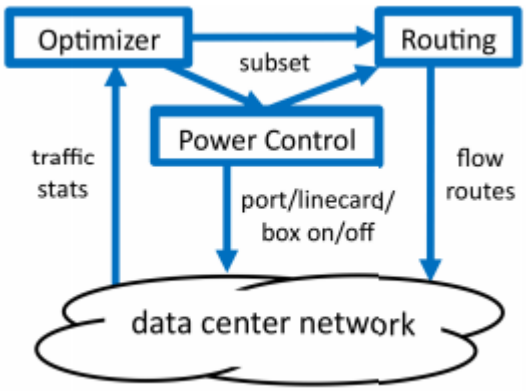

Figure 5: Elastic-tree system diagram

3.5 POWER CONSUMPTION MODEL

Since elastic tree calculates the minimum power network we need a model to determine the power consumption of a network. The model used is as proposed in *"A power benchmarking framework for network devices"* [19].
The power consumption of a switch can be decomposed in the power consumption of a fixed component that includes the chassis power and the power for each linecard and a variable component that depends on the number of active ports, their capacity and utilization.
The power consumption of a switch in defined by the following formula:

$$P_{switch} = P_{chassis} + n_{linecards} * P_{linecards} + \sum_{i} n_{portr_i} * P_{r_i} * F_{r_i}$$

Where:

- P_{switch} is the power consumption by a switch.
- $P_{chassis}$ is the power consumption by a chassis.
- $n_{linecards}$ is the number of linecards.
- $P_{linecards}$ is the power consumption by a linecard without active ports.
- n_{portr_i} is the number of active ports running at rate r_i
- F_{r_i} is an utilization scaling factor for each port, we consider it identical for all switches
- $* P_{r_i}$ is the power consumption of a port at rate r_i

From this we can observe that a switch still consumes energy even when no traffic goes through it. This is what motivates elastic-tree to find a subset network that minimizes the the total power consumed, which corresponds with the summatory of the power consumption of all active switches calculated as described above.

3.6 TRAFFIC

To better evaluate the energetic saving derived from using the protected elastic tree topology we consider three types of traffic that will be evaluated both separately and jointly. Figures 6,7,8 show examples of each kind of traffic.

- Near traffic: refers to traffic flows whose source and destination servers are connected to the same edge switch.

- Middle traffic: refers to the traffic flow whose source and destination servers are linked to the same POD but to different edge switches. The route that a middle traffic flow would follow would start at the source server, going up to the edge switch it is linked to, the pass by an intermediate aggregate switch inside the POD

to return to the edge layer, specifically to the edge switch connected to the destination server where the transmission would finish.

- Far traffic: refers to the traffic flow whose source and destination servers belong to different PODs in the data center. The route of a far traffic flow would go from the source server to the destination server passing through an edge and an aggregation switch in the source POD, then a core node that would route the flow through an aggregation and edge switches in the destination POD and finally it would arrive to the destination server.

Results for mixed traffic will also be computed, this results will contain random traffic requests for all near, middle and far traffic.

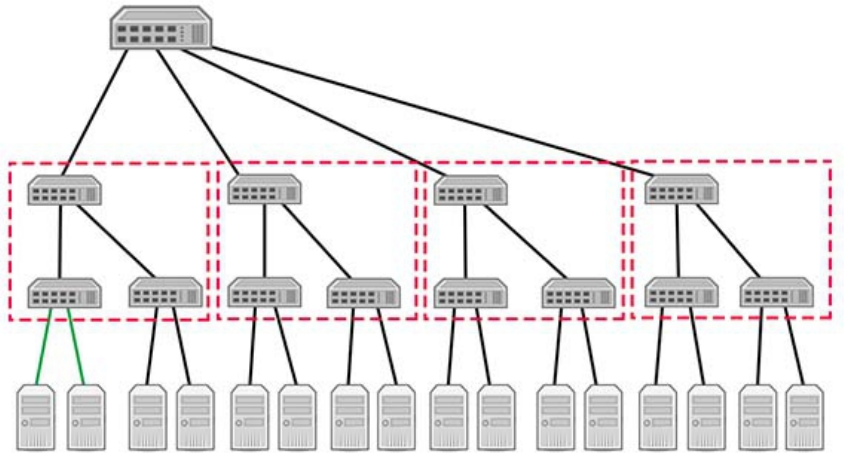

Figure 6: Example of near traffic

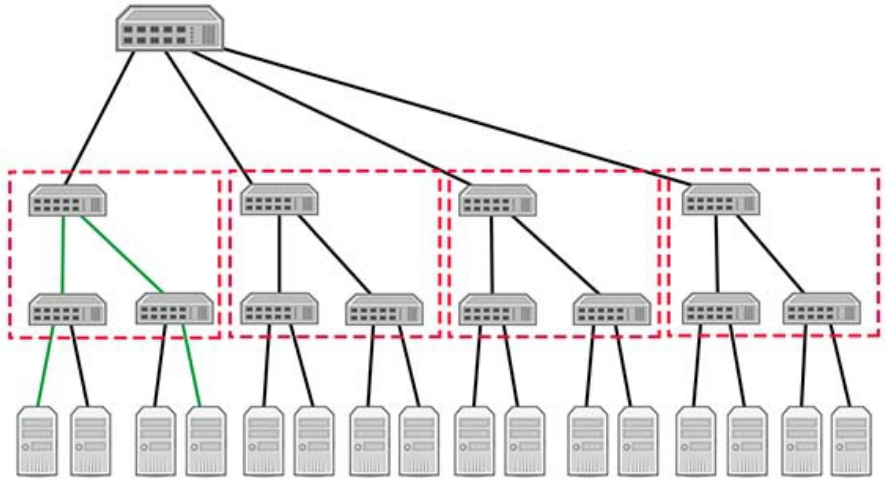

Figure 7: Example of Middle traffic

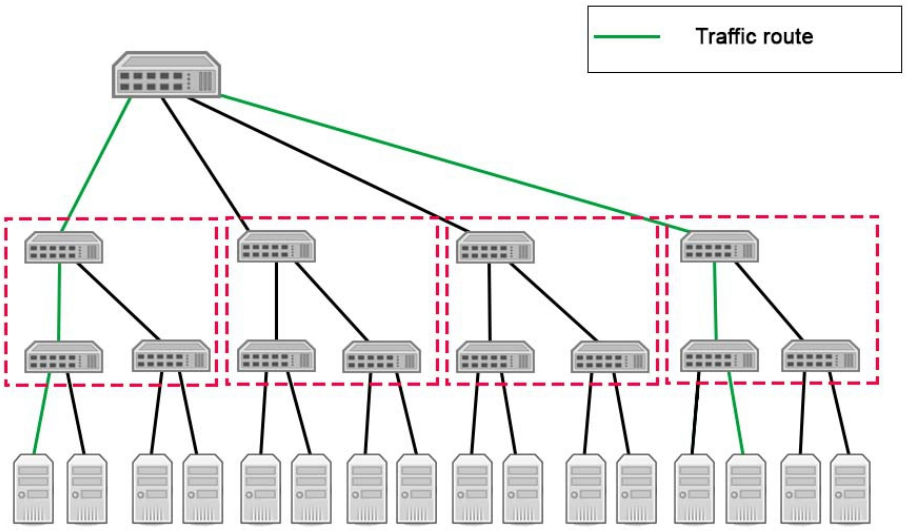

Figure 8: Example of far traffic

3.7 TAH

Topology aware heuristics (TAH)[5] is one of the methods developed to compute a minimum-power network subset in elastic tree. As previously introduced, TAH calculates the necessary components in the network to allocate all traffic demand.

TAH is a simple method that computes the subset with less information than other methods and, therefore, in a fraction of the time. It does not compute the flow rutes, and assumes perfectly divisible flows.
TAH seeks to identify the number of necessary active switches instead of finding which specific switches. The algorithm does this following a simple principle, the number of required switches in a layer is equal to the number of active links required to support the traffic of the most active source above or below.

DETAILED DESCRIPTION OF THE ALGORITHM

Topology-Aware Heuristic Calculates first the number of active links and the uses the obtained results to calculate the number of minimum necessary switches.
We compute the links as follows:

The first step is to compute the minimum number of active links needed to satisfy traffic exiting edge switch $e$in pod p to support up-traffic (edge → agg).

$$LEdge_{p,e}^{up} = \lceil (\sum_{a \in A_p} F(e \to a))/r \rceil$$

Where:

- $LEdge^{up}$ is the number of links needed for traffic flow from edge switch e to aggregation switch a.
- $A_p$is the set of aggregation switches in pod p
- $F(e \to a)$ is the traffic flow from switch e to aggregation switch a
- r is the link rate

In the same manner we calculate the number of links needed to support the down traffic for edge switch e .

$$LEdge_{p,e}^{down} = \lceil (\sum_{a \in A_p} F(a \to e))/r \rceil$$

To obtain the minimum number of links for edge switch e in pod p we compute the maximum of the values previously calculated for up and down traffic.

$$LEdge_{p,e} = \max\{LEdge_{p,e}^{up}, LEdge_{p,e}^{down}, 1\}$$

We obtain the minimum links exiting each edge switch in the network and then move on to calculating the minimum amount of active links needed from each pod to the core. Being

$LAgg^{up}$ the minimum number of links from pod p to the core to satisfy up-traffic, we compute:

$$LAgg_p^{up} = \lceil (\sum_{c \in C, a \in A_p, a \to c} F(a \to c))/r \rceil$$

Just like previously with the links exiting an edge node we calculate the links for down traffic between core and pod p.

$$LAgg_p^{down} = \lceil (\sum_{c \in C, a \in A_p, c \to a} F(c \to a))/r \rceil$$

The maximum of these values is the minimum number of core links for p:

$$LAgg_p = \max\{LEdge_p^{up}, LEdge_p^{down}\}$$

Once we have computed all links we use the results to calculate the number of switches: All edge switches need to be active to connect all servers in the network. Therefore, we only need to calculate the number of active switches in aggregation and core layers. We can use the number of links to calculate the number of switches since every active link must connect two active switches.
We start by considering the switches in the aggregation layer to satisfy up-traffic (edge →agg) in pod p:

$$NAgg_p^{up} = \max_{e \in E_p}\{LEdge_{p,e}^{up}\}$$

And then we compute the minimum number of aggregation switches required to support down-traffic (core → agg) in pod p:

$$NAgg_p^{down} = \lceil (LAgg_p^{down}/(k/2) \rceil$$

Where k is the switch degree.
As with the number of links, the minimum number of active aggregation switches is the maximum between $NAgg_p^{down}$ and $NAgg_p^{up}$:

$$NAgg_p = \max\{NAgg_p^{up}, NAgg_p^{down}, 1\}$$

Finally we need to calculate the number of active core switches. We will need as many cores as links has the most active pod. Therefore:

$$NCore = \lceil \max_{p \in P}(LAgg_p^{up}) \rceil$$

CHAPTER 4: PROPOSED SOLUTION

In this chapter you can find all information related to the algorithms that are developed in this thesis.

4.1 MINIMUM ACTIVE TOPOLOGY WITH SINGLE FAILURE PATH PROTECTION

Once we have implemented the TAH algorithm we move on to consider how to protect the network in case of a single failure. In this section we present all information regarding the algorithm that has developed to protect from single path failure.
The minimum active topology with path protection is a subnetwork that can accommodate all traffic requests while providing a backup for each flow.
We start by investigating the different situations that one single failure can cause in the system and what would be the minimum active extra elements needed to provide a backup path. Three scenarios have been found:

- Failure between core and aggregation layers. Figure 10 shows this scenario and the elements needed for backup with a far traffic request. In case of a failure between a core and aggregation switch we can still find a path between the servers activating one other core node that is connected to the aggregation links in the original working path. A failure between this layers would have no effect for middle or near traffic requests.

- Failure between aggregation and edge layers. A failure between an aggregation and an edge node is the worst case scenario. With far traffic to find a new path between the servers we would have to activate two aggregation switches, one in the source and one in the destination pod as well as a core node connecting to these. In case of middle traffic we would only need to activate one extra aggregation switch. Figure 11 shows the working and backup flow in case of an aggregation-edge failure for far traffic.

- Failure between edge layer and the servers. Our algorithm does not protect against failures between edge nodes and servers due to the fact that the fat tree topology employed offers only one connection per server, which means that a failure in such link would make the server impossible to reach.

We can conclude that to protect from every kind of failure it is necessary to implement the algorithm to activate the minimum number of active elements for the backup path in the

worst case scenario. This would mean to obtain a backup path whose aggregation switches differ from the ones used by the working path and consequently with a different core node.

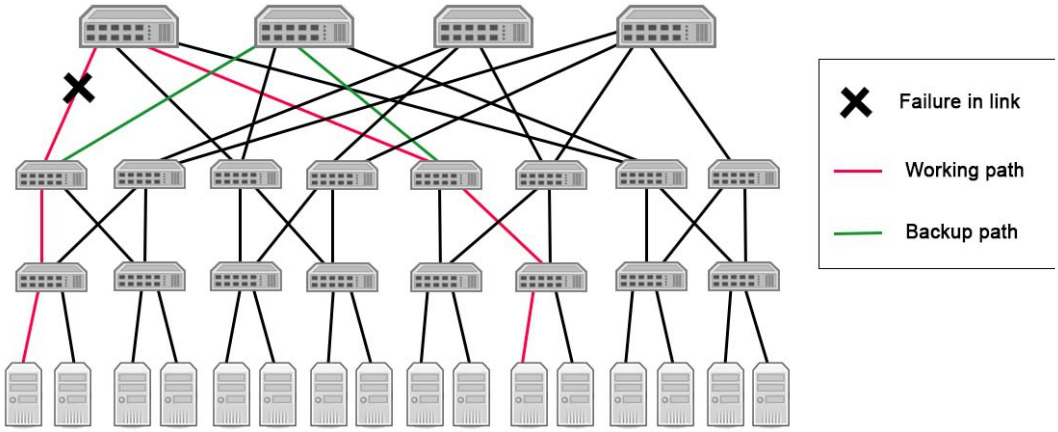

Figure 10: example of working and backup paths for core-aggregation failure

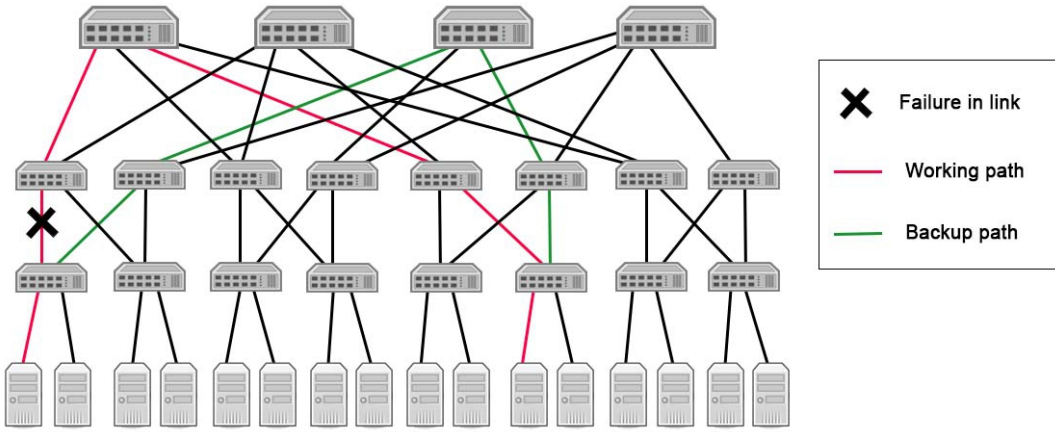

Figure 11: example of working and backup paths for aggregation-edge failure

4.1.1 DETAILED DESCRIPTION OF THE ALGORITHM

The algorithm finds a minimum active topology with path protection mechanism with the assumption that we know a rough traffic matrix of the traffic between edge switches since we focus on the protection between them instead of between servers.
In order to find the minimal network subset, we will try to accommodate working and backup paths for all flows in the traffic matrix. We will activate switches and links gradually as we need them.
We have to follow several steps to allocate all flows:

- The algorithm starts with an initial active topology that corresponds to the Minimum spanning tree (MST) of the fat tree topology, where the root is one core node and the servers are the leaves.

- We begin to allocate all working paths for the flows included in the matrix. To do so we browse all active paths in the network that connect the source and the destination edge switches e_s and e_d to find the one with the largest residual bandwidth. In the case that the active paths have no residual bandwidth and therefore cannot accommodate the traffic flow, we browse all possible paths including the inactive elements to find one that requires least additional power consumption to activate the elements in it. The power consumption is calculated as in TAH and how it was presented in section 3.5 of this thesis.

- Once we have all working paths we move on to allocating the backup paths for each of them. To do so, we follow a similar process than with the working paths. We first browse all active paths in the network that connect the source and destination edge switches to find the one with largest residual bandwidth. The difference this time is that we have to exclude the paths containing the aggregation switches present in the working path to ensure that backup and working path are completely disjoint. In case of not finding a path with available bandwidth for the flow, just like when allocating the working flows, we browse all paths including inactive elements to find the one that requires least additional power still excluding the aggregation switches from the working flow.

4.1.2 IMPLEMENTATION

INPUT

Below an explanation of the input parameters to the algorithm in order to obtain the minimum active topology:

- k: is the fat tree degree necessary to compute the network structure. With it we calculate:
    - number of core switches ($(k/2)^2$)
    - number of pod (k)
    - number of aggregation switch per pod (k/2)
    - number of edge switches per pod (k/2)
    - number of servers: ($k^3/4$)
    - number of ports per switch (k)

- c: indicates the link capacity. We assume all links have an identical capacity.
- p: fixed part of a switch's power consumption.
- pport: power consumption of one active port.
- mat: traffic matrix containing all traffic requests between servers. In it we can find a list of all traffic requests represented by three parameters:
    - s: server source ID
    - d: server destination ID
    - b: requested bandwidth in Gbps

OUTPUT

The algorithm will output the minimum active topology as well as statistics to later analyze the results more easily:

- Active elements in the minimum active topology: lists all switches and elements that need to be active in order to satisfy the given traffic.
    - Active core switches
    - Active aggregation switches
    - Active edge switches
    - Active links

- Statistics:
    - Number of active switches
    - Number of active ports: calculated as twice the active links since each link connects two ports in different switches.
    - Power consumption: the power that the active part of the network would consume. Calculated as detailed in section 3.5 of this paper.
    - Utilisation index: represents the data center's load. It is defined as the ratio between the total requested bandwidth between the maximum acceptable load.

PSEUDOCODE

Following the description of the algorithm that can be found in the previous section a pseudocode has been generated to later help with the implementation of the algorithm:

```
//Allocation of working path
    MaxResCap=0;
    MinPow=∞;
    For each flow f ∈FlowMatrix
        For each path p∈actPath between e_s and e_d
            if(p.capacity > MaxResCap)
                Max = p.capacity
                workingPath = p
        if(MaxResCap==0)
            For each path p' ∈allPath between e_s and e_d
                if(p'.power < MinPow)
                    MinPow = p.power
                    workingPath = p
            activate(workingPath)
        workingPath.capacity -= f.bandwidth
    //Allocation of backup path
    MaxResCap=0;
    MinPow=∞;
    For each flow f ∈FlowMatrix
        For each path p∈actPath between e_s and e_d where workingPath.a_s ∉ p and
        where workingPath.a_d ∉ p
            if(p.capacity > MaxResCap)
                Max = p.capacity
                BackupPath = p
        if(MaxResCap==0)
            For each path p' ∈allPath between e_s and e_d where workingPath.a_s ∉ p
            and where workingPath.a_d ∉ p
                if(p'.power < MinPow)
                    MinPow = p.power
                    BackupPath = p
            activate(BackupPath)
    If (!workingPath) return false;
```

FUNCTIONS

To ease the code and make it more modular the work of the algorithm has been divided in several methods that can be found in table 3 together with a brief description.

struct(double,list) path(co,s,j,cap)	Finds the active path from core co to the edge switch s with the greatest residual bandwidth. Returns the path and its capacity.
struct(double,list) Bpath(co,s,j,cap,ag)	Finds the active path connecting the core switch co and the edge switch s that does not contain the aggregation switch ag with the greatest residual bandwidth.
list reverse(pcs)	Auxiliar function to change the format of the path from the core to the destination edge switch.
void refreshCap(t,path,b,i)	Once found a working path for a flow it changes decreases the capacity of the path to allocate the bandwidth demand of the flow.
struct(double,double,list) actPath(co,s,j,cap)	In case of not finding an active path actPath browses all possible paths between core switch co and edge switch s to find one with minimum additional power consumption.
struct(double,double,list) actBPath(co,s,j,cap,ag)	In case of not finding an active path actPath browses all possible paths between core switch co and edge switch s to find one with minimum additional power consumption that does not contain aggregation switch ag.
void activatePath(tree,path,i)	Activates all elements that are not in path "path"
struct(double,list) findpath(s,d,b,tree)	Main function of the algorithm in charge of allocation the working path.
struct(double,list) findBackupPath(s,d,b,tree,pt)	Main function of the algorithm in charge of allocating the backup path

Table 3: functions in single failure algorithm

4.2 MINIMUM ACTIVE TOPOLOGY WITH DOUBLE FAILURE PATH PROTECTION

To get an even deeper understanding of how much protection we can trade for energy efficiency while still significantly improving energy consumption to the fat-tree topology it's been decided to also consider the scenario where there are two failures in the system.
All relevant scenarios caused by double failure have to be considered together with the additional elements that should be active to protect the network in each case. We consider the one failure scenarios already protected by one backup path.

- Two failures in core-aggregation links
    - Two failures in links exiting the same core node. The relevant cases are when the down links are both in the working path or in the backup path. Figure 12 illustrates these scenarios. In case that both failures are in the working path we can still route the traffic through the backup path. Similarly if both failures are in links that are part of the backup path we will route the traffic through the working path as usual.
    - Two failures in links exiting different core nodes. Having two failures in links exiting different cores entails a more challenging scenario. We may have a failure in one of the links exiting one core that connect to either the source or destination pod of the working path and one other in a link exiting a different core c2 that is part of the backup path. In this case neither the working path nor the backup path can be used so we need to activate one extra core connecting the two aggregation switches of the working path. Figure 13 illustrates this scenario.

- Two failures in aggregation-edge links
    - Two failures in the working path. Having two failures in the working path for aggregation-edge links means that the links that the links of the working path for both source and destination pods are down. In this case we can still route the traffic through the backup path.
    - Two failures in the backup path. In the same manner having two links down in the source and destination pod of the backup path does not pose any problem since the working path is still active in its entirety.
    - One failure in the working path and one in the backup path. If we have links down in both the working and backup paths inside the same or different pods the servers are completely disconnected for k=4 and there is no way to protect for this kind of error. For topologies with higher degrees we can find a second backup path connecting the source and destination edge switches.

Figure 14 illustrates this type of failure for k=4 and figure 15 does the same for k=6.

- One failure in a core-aggregation link and one in an aggregation-edge link
    - Two failures in the working or in the backup path. Having two failures, one in core-aggregation and one in aggregation-edge would only be relevant if both links are part of the working path or of the backup path but, like previously, in either of those cases the path with no failures can be used.
    - One failure in the working path and one in the backup path. Two cases should be considered here. The first one is when the core-aggregation failure is part of the working path and the aggregation-edge is part of the backup path. In this case to protect from failure it would be necessary to activate a core switch that connects the aggregation switches of the working path. The second case is the opposite, being the core-aggregation failure part of the backup path and the aggregation-edge part of the working path. If this is the case, we shall activate one core node that will connect the aggregation switches in the backup path. Figures 16 and 17 show these scenarios.

As it happened with single failure, to ensure that we protect from every type of double failure we should activate as many extra elements as in the worst case scenario. In this case that would be having two failures in aggregation-edge links being the links one in the working path and one in the backup path needing two backup paths to ensure all communications are satisfied. By doing so we can no longer ensure protection for every network, for 4-ary fat-trees it is only possible to find one backup path between two given edge switches. This means that for the double failure protection algorithm we will assume that the fat tree has $k \geq 6$.

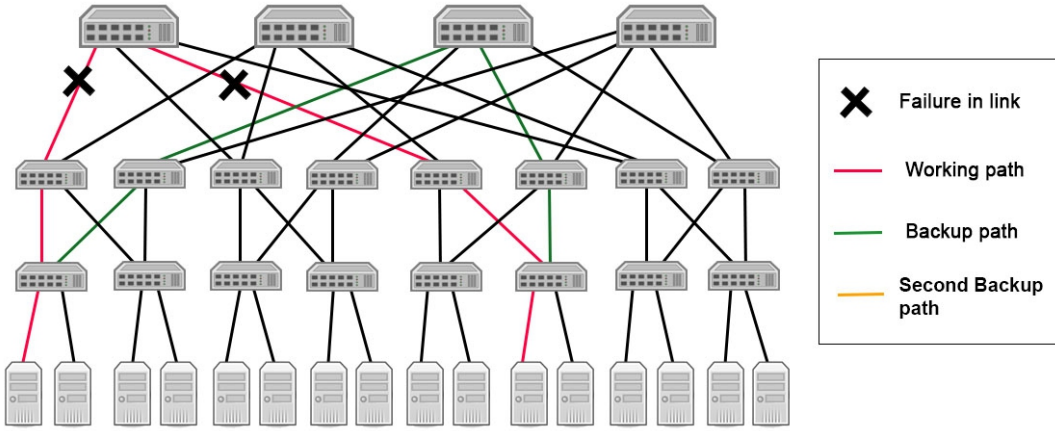

Figure 12: two failures between the core and aggregation layers affecting the working path

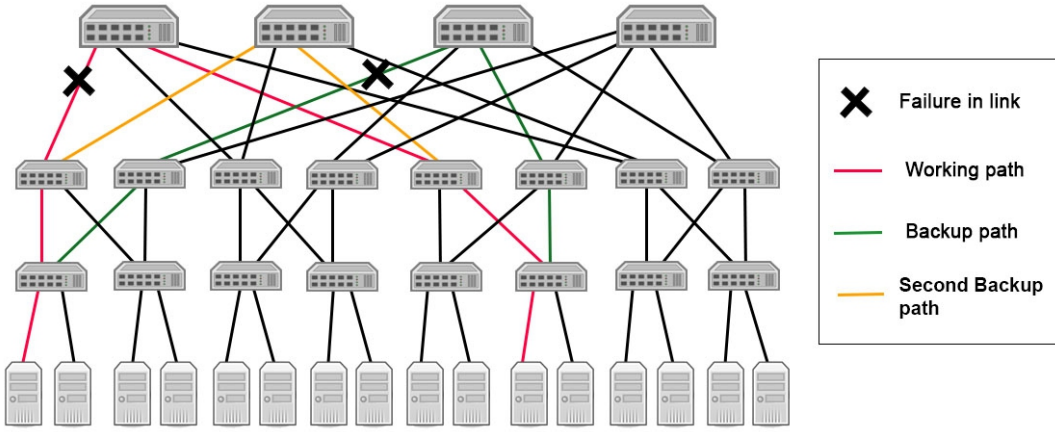

Figure 13: two failures between the core and aggregation layers affecting both the working and backup paths

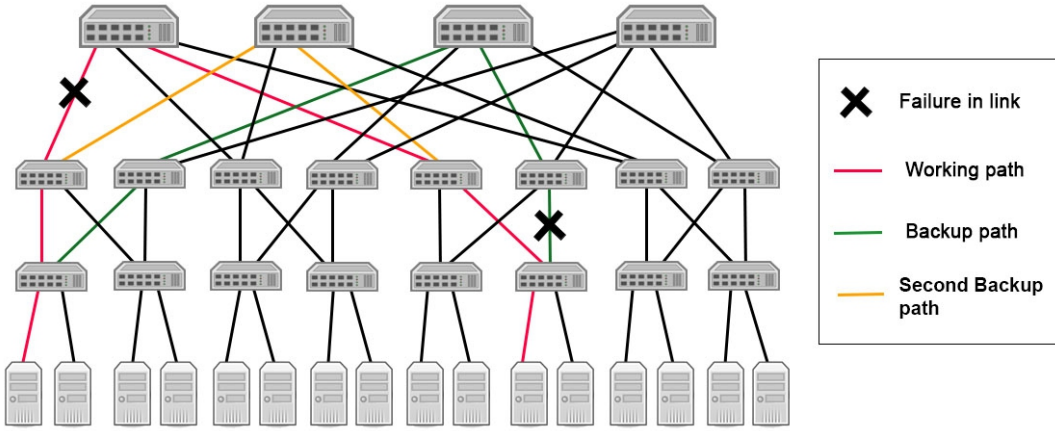

Figure 14: One failure between the core and aggregation layer and another between aggregation and edge the first affecting the working path and the second the backup path.

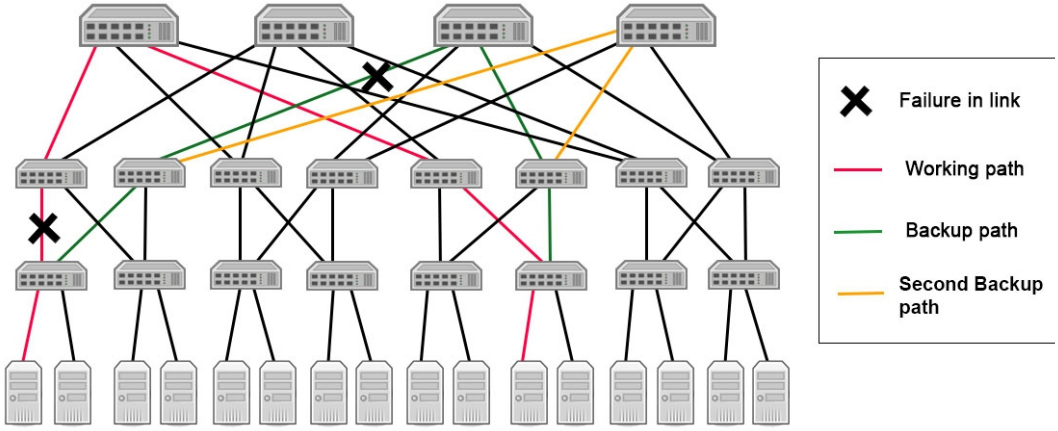

Figure 15: One failure between the core and aggregation layer and another between aggregation and edge the first affecting the backup path and the second the working path.

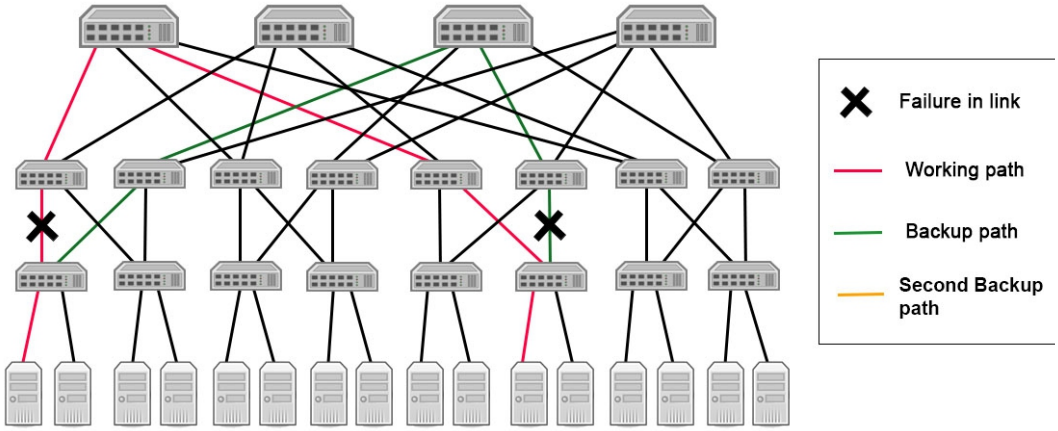

Figure 16: Scenario with two failures between the aggregation and edge layers affecting both working and backup path for k=4.

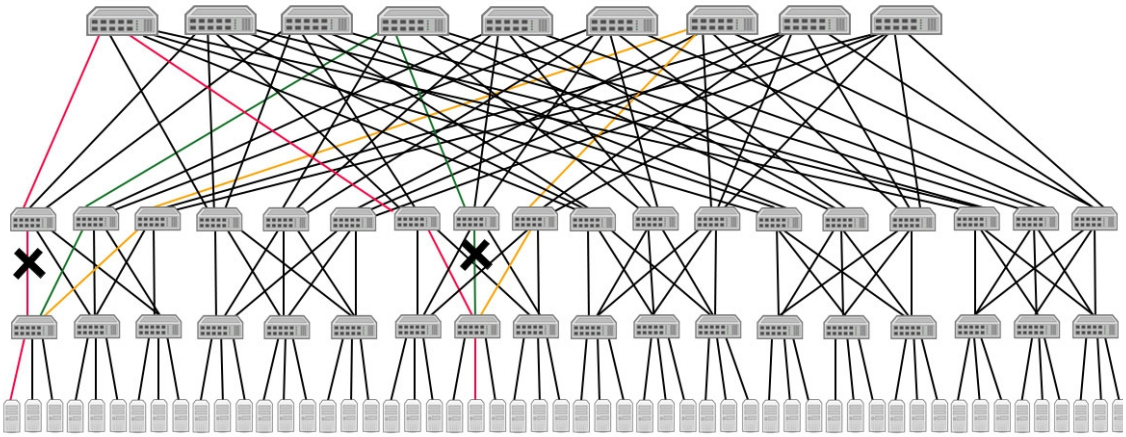

Figure 17: Scenario with two failures between the aggregation and edge layers affecting both working and backup path for k=6.

4.2.1 DETAILED DESCRIPTION OF THE ALGORITHM

The algorithm designed to obtain a minimum active topology with double failure path protection works practically identically to the single failure algorithm. The difference is that after allocating all working paths and backup paths we find a second backup path for every traffic flow in the matrix. Therefore we follow the next steps:
We have to follow several steps to allocate all flows:

- The algorithm starts with an initial active topology that corresponds to the Minimum spanning tree (MST) of the fat tree topology, where the root is one core node and the servers are the leaves.

- We begin to allocate all working paths for the flows included in the matrix.
- Once we have all working paths we move on to allocating the backup paths for each of them.
- Finally we allocate the second backup path for every flow in the matrix. To do so we start by browsing all paths containing only active elements that connect the source edge switch $e_s$and the destination edge switch e_d. We exclude the aggregation switches in both the working and backup paths. We try to find the path with largest residual bandwidth. In case of all active paths that satisfy the conditions we move on to browsing all paths in the network connecting e_s and e_d that do not contain any of the excluded aggregation switches. We select as second backup path the path that requires the minimum additional power consumption.

4.2.2 IMPLEMENTATION

INPUT

We maintain the same input used for the minimum active topology with single failure path protection algorithm.

OUTPUT

The output follows the same format as the one used in the minimum active topology with single failure path protection algorithm.

PSEUDOCODE

The pseudocode has been generated from the pseudocode in the single failure algorithm adding the search of a second backup path.

```
//Allocation of working path
    MaxResCap=0;
    MinPow=∞;
    For each flow f ∈FlowMatrix
        For each path p∈actPath between e_s and e_d
            if(p.capacity > MaxResCap)
                Max = p.capacity
                workingPath = p
        if(MaxResCap==0)
            For each path p' ∈allPath between e_s and e_d
                if(p'.power < MinPow)
```

MinPow = p.power
workingPath = p
activate(workingPath)
workingPath.capacity -= f.bandwidth
//Allocation of backup path
MaxResCap=0;
MinPow=∞;
For each flow f ∈FlowMatrix
For each path p∈actPath between e_s and e_d where workingPath.a_s∉p and where workingPath.a_d
if(p.capacity > MaxResCap)
Max = p.capacity
BackupPath = p
if(MaxResCap==0)
For each path p' ∈allPath between e_s and e_d where workingPath.a_s∉p and where workingPath.a_d
if(p'.power < MinPow)
MinPow = p.power
BackupPath = p
activate(BackupPath)
//Allocation of second backup path
MaxResCap=0;
MinPow=∞;
For each flow f ∈FlowMatrix
For each path p∈actPath between e_s and e_d where workingPath.a_s∉p and workingPath.a_d ∉p and BackupPath.a_s∉p and BackupPath.a_d ∉p
if(p.capacity > MaxResCap)
Max = p.capacity
BackupPath2 = p
if(MaxResCap==0)
For each path p' ∈allPath between e_s and e_d where workingPath.a_s∉p and workingPath.a_d∉p and BackupPath.a_s∉p and BackupPath.a_d ∉p
if(p'.power < MinPow)
MinPow = p.power
BackupPath2 = p
activate(BackupPath2)
If (!workingPath) return false;

FUNCTIONS

Most of the functions are maintained from the single failure algorithm. Therefore this section will only introduce the new functions added in order to protect also from double failure.
Table 4 contains the new functions as well as a brief explanation.

struct(double,list) Bpath2(co,s,j,cap,ag,ag2)	Finds active path connecting core switch co and edge switch s that does not contain the aggregation switches in the working path or in the backup path. Returns the path with maximum capacity and the capacity itself.
struct(double,double,list) actBpath2(co,s,j,cap,ag,ag2)	In case of not finding an active path, actBPath2 finds a path between core switch co and edge switch s that does not contain any of the aggregation switches in the working or in the backup path. Returns the path with minimum additional power consumption and the power consumption itself.
struct(double,list) findBackPath2(s,d,b,tree,pt,ptb)	Main function to allocate the second backup path, it

Table 4: functions added to protect from double failure.

4.3 FAT-TREE TOPOLOGY

The fat tree structure used in all three versions of the algorithm is been implemented in java.
Scilab was the language chosen to implement the algorithms but its structures are rather limited. One advantage is that scilab is compatible with java code and allows to use objects in this language.
The following section will expose how the structure was designed and later implemented as well as the methods used in scilab to be able to use the code generated in java.

4.3.1 IMPLEMENTATION

Figure 9 shows the UML design of the fat tree including its attributes. It is a very simple design based in the principle that the fat tree is just a set of nodes. The class fat tree acts as

an invisible core node due to the fact that the fat tree that we use has several and not just one.

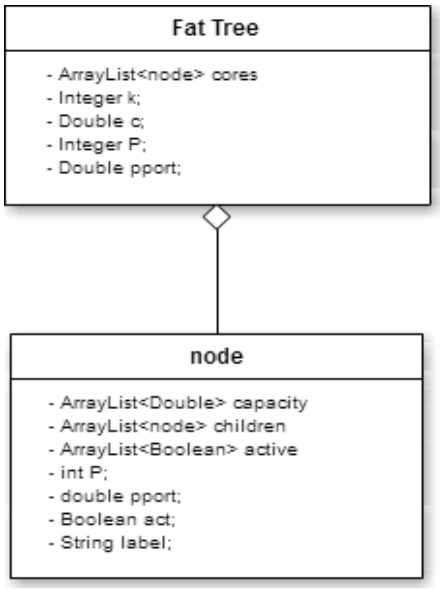

Figure 9: UML design of the fat tree structure

Here is a brief explanation of the parameters in each class:

- FatTree:
  - ArrayList<nodes> cores: list of the $(k/2)^2$ cores in the network
  - Integer k: degree of the fat tree used to calculate the parameters as presented in the input section generate the tree
  - Double c: capacity of each link
  - Integer P: fixed part of the switch's power consumption
  - Double pport: power consumption of one active port
- node:
  - ArrayList<Double> capacity: list of the capacity available in every link connecting the node to its children.
  - ArrayList<node> children: list of all children nodes, those connected to the node in an inferior layer if any.
  - ArrayList<Boolean> indicates which of the children are active.
  - int P: fixed part of the switch's power consumption
  - double pport: power consumption of one active port
  - Boolean act: indicates if the node is active
  - String label: identifier of the node, it is composed by a letter and a number. The letter indicates if it is a core, edge, aggregation node or server, we use c,e,a and s respectively. The number is to differentiate between switches in a same layer or servers.

Each class has a set of methods that are used by the algorithms, tables 1 and 2 show the methods, the parameters that are needed to invoke them and a description of fatTree and node class respectively.

FatTree(Integer kk, Double cc, Integer pp, double pportt)	Creator method, generates the tree with the parameters given by the user.
void activate(String s)	Activates a node with label s.
void activate(String par, String chi)	Activates a link between par and chi as well as the nodes themselves if they are not already active.
node findCore(String s)	Returns the core node in the network whose label is s.
node findNode(String s)	Returns the node in the network whose label is s.
String getAgg()	Returns the labels of all aggregation switches that are active
Double getCapacity()	Returns the value of c, the capacity of all links in the network.
String getCore()	Returns a string containing all the labels of active core nodes.
String getEdg()	Returns a string containing all the labels of active edge nodes.
Integer getK()	Returns the value of k, the degree of the tree.
String getLinks()	Returns a string containing all active links in the tree.
Integer getNCore()	Returns the number of core nodes in the network.
Integer getPower()	Returns the value of p.
double getPport()	Returns the value of pport.
int NAgg()	Returns the number of aggregation switches that are active.
int NAcore()	Returns the number of core switches that are active.
int NAEd()	Returns the number of edge switches

	that are active
int NAlink()	Returns the number of links that are active.
void printTree()	Prints the tree in text format indicating nodes and their connections.
void setCapacity(Double cc)	Sets the value of c to cc.
void setPower(Integer pp)	Sets the value of P to pp.
void setPport(double port)	sets the value of pport to port.

Table 1: methods in FatTree class

node(int pp, double pport, Boolean a, String l)	Creator method. Generates a node with the parameters given
void activate (String s)	Activates the children node whose label is s.
void addchild(node n, Double cap, Boolean b)	Adds node n to the list of children, cap indicates the capacity of the link connecting to node n and boolean b if it is active.
void changeCap(String s, Double cap)	Decreases the capacity available of the link connecting to the node with label s.
void copy(node n)	Generates a copy of the node n.
void copychild(node n)	The children of the current node become the same that node n has.
void deactivateAll()	Deactivates all links connecting to the children nodes.
node findNode(String s)	Returns the node whose label is s.
Boolean getAct()	Returns the value of act.
ArrayList<Boolean> getActive()	Returns the list that indicates which links are active.
Boolean getActive(int i)	Indicates if the link connecting to the i^{th} child is active.
ArrayList<Double> getCapacity()	Returns the list of link capacities.

Double getCapacity(int i)	Returns the capacity of the link connecting to the i^{ti} child.
node getChild(int i)	Returns the i^{th} child.
ArrayList<node> getChildren()	Returns a list of all children.
String getLabel()	Returns the value of the node's label.
Integer getNact()	Returns the number of active links exiting the node.
Integer getNChild()	Returns the number of children the node has.
int getP()	Returns the value of P.
double getPport()	Returns the value of pport.
Boolean hasChildren()	Indicates if the node is a leaf or if it has children.
void printNode()	Prints the node's information in text format.
void setAct(Boolean b)	Sets the act attribute to b.
void setActive(ArrayList<Boolean> act)	sets the list of active links to act.
void setActive(Boolean b, int i)	Sets the link to the i^{th} child to b.
void setCapacity(ArrayList<Double> c)	Sets the list of link capacities to c.
void setCapacity(Double c, int i)	Sets the link capacity to the i^{th} child to c.
void setChildren(ArrayList<node> ch)	Sets the list of children to ch.
void setLabel(String l)	Sets the node's label to l.
void setP(int p)	Sets the value of P to p.
void setPport(double pport)	Sets the value of pport to pport.

Table 2: methods of the class node

4.3.2 JAVA OBJECTS IN SCILAB

Scilab can invoke java objects thanks to an additional package given the name of Java interaction mechanism in scilab (JIMS)[20]. The start guide written by Simon Billemont presents how to install and began to use the package as well as the instructions needed to use the java objects and several examples.
JIMS has been vital to make compatible the fat tree code generated in java and the algorithms, implemented in scilab. Here we will present some of the methods and commands that have been necessary in the development of this thesis.

Javaclasspath: javaclasspath tells scilab where the complied class information is stored in the computer. It works simply by giving the path as parameter of the method. In this case:

"javaclasspath("C:\Users\Elena\Documents\NetBeansProjects\FatTree\dist\FatTree.jar");"

jimport: imports a java class into the workspace. to use jimport we follow the next structure:
jimport('packageName.ClassName') we needed to import both fatTree and node so we use:

```
jimport('fattree.FatTree');
jimport('fattree.node');
```

This is equivalent to a class object in java, now we can generate objects from the classes. We can do so with the following instruction:

```
tree = FatTree.new(k,c,p,pport);
```

We will have an object from the FatTree class named tree with the parameters k,c,p,pport.

jinvoke: this instruction lets us invoke a method inside a java class to be used in scilab. the structure used for jinvoke is: jinvoke(object,'method', 'paramater1', 'parameter2'...) we could also use invoke typing object.method(parameters).

JIMS has many other more complex functionalities but to keep the code as clean and easy as possible we have only used the methods described above.

CHAPTER 5: EXPERIMENTS

5.1 EXPERIMENTAL SCENARIOS

In this section some experimental scenarios are presented. We will inspect the results obtained from several inputs with different conditions and study if the minimum topology obtained in each case protects the data center against failures as expected.

5.1.1 EXPERIMENTAL SCENARIOS FOR SINGLE PATH PROTECTION

For the algorithm of minimum active topology with single path protection we will evaluate three different scenarios. To simplify, the scenarios selected use the minimum required degree of fat-tree structures (k=4) to protect against single failure. The traffic requests for each of the scenarios has been randomly generated for far, middle and mixed traffics. The code used to generate the inputs for each of the scenarios can be found in the annex.

A) FIRST SCENARIO: SINGLE FAILURE PROTECTION WITH FAR TRAFFIC

INPUT PARAMETERS

The value of the input parameters for the first scenario is as follows:

- fat-tree degree (k): 4.
- Link capacity (c): 10.
- Switch power consumption (P): 146.
- Port power consumption (pport): 0.9.
- Number of traffic requests: 10.
- Type of traffic: Far traffic.
- Traffic matrix:

Source server	Destination server	Requested bandwidth (Gbps)
s15	s1	0.79
s11	s5	0.19
s15	s8	0.84
s1	s9	0.97

s2	s10	0.64
s8	s9	0.81
s6	s11	0.09
s1	s8	0.64
s2	s14	0.64
s9	s14	0.92

OUTPUT

- Active core switches: c1, c3
- Active aggregation switches: a1,a2,a3,a4,a5,a7,a8
- Active edge switches: e1,e2,e3,e4,e5,e6,e7,e8
- Active links: c1-a1, c1-a3, c1-a5, c1-a7, c3-a2, c3-a4, c3-a6, c3-a8, a1-e1, a1-e2, a3-e3, a3-e4, a5-e5, a5-e6, a7-e7, a7-e8, a2-e1, a4-e3, a4-e3, a6-e5, a6-e6, a8-e7, a8-e8.
- Number of active switches: 18
- Number of active ports: 46
- Power consumption: 2669.4
- Utilisation factor: 8.1625%

PROTECTION AGAINST SINGLE FAILURE

As previously stated in section 3.6 of this paper there are two relevant scenarios with single failure in which our algorithm will protect the data center, One when the failure is in one link connecting the core and aggregation layers that is part of the working flow of at least one traffic request and the other one is a failure in a link connecting switches in the aggregation and edge layers that as in the previous case is part of a working path.

If a failure were to affect the link c1-a5, for example, several traffic requests would need to be routed through a different path. Table 5 shows all the affected communications, their working path and their backup path. As we can see all the elements both in the working and in the backup path are active in the minimum active topology that has been obtained from the algorithm. This means that even if the working paths cannot be used there exists another path that can still be used, being the system protected for this kind of failure.

Traffic request	Working path	Backup path
s11->s5, 0.19Gbps	s11-e6-a5-c1-a3-e3-s5	s11-e6-a6-c3-a4-e3-s5
s1->s9, 0.97 Gpbs	s1-e1-a1-c1-a5-e5-s9	s1-e1-a2-c3-a6-e5-s9
s2->s10, 0.64 Gbps	s2-e2-a1-c1-a5-e5-s10	s2-e2-a2-c3-a6-e5-s10
s8->s9, 0.81 Gbps	s8-e4-a3-c1-a5-e5-s9	s8-e4-a4-c3-a6-e5-s9
s6->s11, 0.09 Gbps	s6-e3-a3-c1-a5-e5-s10	s6-e3-a4-c3-a6-e5-s10

Table 5: working and backup paths of traffic requests affected by a failure in core-aggregation

The same can be done with any link connecting switches from the aggregation and the edge layer:
If the link connecting a3 and e3 for example were to malfunction two of the traffic requests would be affected. Table 6 shows their working and backup path. As before all elements were active in the topology obtained from the algorithm and are therefore protected from single failure. This could be done for any relevant link in the network with similar results.

Traffic request	Working path	Backup path
s11->s5, 0.19 Gbps	s11-e6-a5-c1-a3-e3-s5	s11-e6-a6-c3-a4-e3-s5
s6->s11, 0.09 Gbps	s6-e3-a3-c1-a5-e6-s11	s6-e3-a4-c3-a6-e6-s11

Table 6: working and backup path of traffic requested affected by single failure in aggregation-edge.

B) SECOND SCENARIO: SINGLE FAILURE PROTECTION WITH MIDDLE TRAFFIC

INPUT PARAMETERS

The value of the input parameters for the second scenario is as follows:

- fat-tree degree (k): 4.
- Link capacity (c): 10.
- Switch power consumption (P): 146.
- Port power consumption (pport): 0.9.
- Number of traffic requests: 10.
- Type of traffic: Middle traffic.
- Traffic matrix: The traffic matrix has been randomly generated. The requested bandwidth is in the range [0,1]. We will not get into detail for the traffic matrix in this scenario.

OUTPUT

- Active core switches: c1
- Active aggregation switches: a1, a2, a3, a4, a5, a7, a8
- Active edge switches: e1, e2, e3, e4, e5, e6, e7, e8
- Active links: c1-a1, c1-a3, c1-a5, c1-a7, a1-e1, a1-e2, a3-e3, a3-e4, a5-e5, a5-e6, a7-e7, a7-e8, a2-e1, a2-e2, a4-e3, a4-e4- a6-e5, a6-e6,a8-e7, a8-e8
- Number of active switches: 17
- Number of active ports: 40
- Power consumption: 2518
- Utilisation factor: 6.4%

PROTECTION AGAINST SINGLE FAILURE

The same comprovations that were done with far traffic can be performed for the results obtained with middle traffic. In this case only failures that occur in links connecting aggregation and edge switches will be relevant since it is not necessary to go through a core node to route middle traffic requests.

C) THIRD SCENARIO: SINGLE FAILURE PROTECTION WITH MIXED TRAFFIC

INPUT PARAMETERS

The value of the input parameters for the third scenario is as follows:

- fat-tree degree (k): 4.
- Link capacity (c): 10.
- Switch power consumption (P): 146.
- Port power consumption (pport): 0.9.
- Number of traffic requests: 10.
- Type of traffic: Mixed traffic.
- Traffic matrix: The traffic matrix has been randomly generated. The requested bandwidth is in the range [0,1]. We will not get into detail for the traffic matrix in this scenario.

OUTPUT

- Active core switches: c1, c3
- Active aggregation switches: a1, a2, a3, a4, a5, a7, a8
- Active edge switches: e1, e2, e3, e4, e5, e6, e7, e8
- Active links: c1-a1, c1-a3, c1-a5, c1-a7, a1-e1, a1-e2, a3-e3, a3-e4, a5-e5, a5-e6, a7-e7, a7-e8, a2-e1, a2-e2, a4-e3, a4-e4- a6-e5, a6-e6,a8-e7
- Number of active switches: 17

- Number of active ports: 40
- Power consumption: 2518
- Utilisation factor: 6.4%

5.1.2 EXPERIMENTAL SCENARIOS FOR DOUBLE PATH PROTECTION

For the algorithm of minimum active topology with double failure path protection we will evaluate three different scenarios. To simplify, as previously, the scenarios selected use the minimum required degree of fat-tree structures (k=6) to protect against double failure.

A) FIRST SCENARIO: DOUBLE FAILURE PROTECTION FOR FAR TRAFFIC

INPUT PARAMETERS

The value of the input parameters for the first scenario is as follows:

- fat-tree degree (k): 6.
- Link capacity (c): 10.
- Switch power consumption (P): 146.
- Port power consumption (pport): 0.9.
- Number of traffic requests: 25.
- Type of traffic: Far traffic.
- Traffic matrix:

Source server	Destination server	Requested bandwidth
s16	s47	0.24
s28	s23	0.88
s18	s9	0.78
s39	s50	0.5
s45	s21	0.99
s8	s45	0.09
s26	s9	0.43
s50	s27	0.35

s11	s22	0.31
s7	s26	0.74
s26	s17	0.55
s37	s52	0.69
s40	s22	0.92
s49	s2	0.04
s11	s1	0.3
s5	s38	0.14
s52	s44	0.18
s29	s8	0.74
s4	s51	0.98
s9	s21	0.13
s22	s36	0.68
s10	s2	0.19
s6	s21	0.12
s50	s43	0.6
s19	s52	0.59

OUTPUT

- Active core switches: c1, c4,c7
- Active aggregation switches: a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11,a12, a13, a14, a15, a16, a17, a18
- Active edge switches: e1, e2, e3, e4, e5, e6, e7, e8, e9, e10, e11, e12, e13, e14, e15, e16, e17, e18
- Active links: c1-a1, c1-a4, c1-a7, c1-a10, c1-a13, c1-a16, c4-a2, c4-a5, c4-a8,c4-a11, c4-a14, c4-a17, c7-a3, c7-a6, c7-a9, c7-a12, c7-a15, c7-a18, a1-e1, a1-e2, a1-e3, a4-e4, a4-e5, a4-e6, a7-e7, a7-e8, a7-e9, a10-e10, a10-e11, a10-e12, a13-e13, a13-e14, a13-e15, a16-e16, a16-e17, a16-e18, a2-e1, a2-e2, a2-e3, a5-e4, a5-e6, a8-e7, a8-e8, a8-e9, a11-e10, a11-e12, a14-e13, a14-e14, a14-e15, a17-e16, a17-e17, a17-e18, a3-e1, a3-e2, a3-e3, a6-e4, a6-e6, a9-e7, a9-e8, a9-e9, a12-e10, a12-e12, a15-e13, a15-e14, a15-e15, a18-e16, a18-e17, a18-e18

- Number of active switches: 39
- Number of active ports: 136
- Power consumption: 5816.4
- Utilisation factor: 4.5%

PROTECTION AGAINST DOUBLE FAILURE

As has been previously stated in this paper there are several scenarios in which our algorithm will protect the data center when double failures occur. Below there is an analysis of the results from the first scenario with double failure using a similar approach to the one in the first single failure scenario.
First we have to identify the cases in which double failure will be relevant and we will need the existence of a second backup path to ensure communications.
We may have two failures in core-aggregation links, two failures in aggregation-edge links or one failure in a core-aggregation and another in an aggregation-edge link. These failures will be relevant in case that one of them pertains to the working path of at least one traffic request and the other to the backup path of the same request.

If we have two failures in core-aggregation links, for example: c1-a1 and c4-a2 some of the traffic requests would experience a double failure scenario where they will be routed through their second backup path. Table 7 shows the affected communications, their backup paths and their backup paths.

Traffic request	Working path	1st Backup path	2nd Backup path
s18->s9, 0.78	s18-e6-a4-c1-a1-e3-s9	s18-e6-a5-c4-a2-e3-s9	s18-e6-a6-c7-a3-e3-s9
s8->s45, 0.09	s8-e3-a1-c1-a13-e15-s45	s8-e3-a2-c4-a14-e15-s45	s8-e3-a3-c7-a15-e15-s45
s26->s9, 0.43	s26-e9-a4-c1-a1-e3-s9	s26-e9-a5-c4-a2-e3-s9	s26-e9-a5-c7-a3-e3-s9
s7->s26, 0.74	s7-e3-a1-c1-a7-e9-s26	s7-e3-a2-c4-a8-e9-s26	s7-e3-a3-c7-a9-e9-s26
s49->s2, 0.04	s49-e17-a16-c1-a1-e1-s2	s49-e17-a17-c4-a2-e1-s2	s49-e17-a18-c7-a3-e1-s2
s11->s1, 0.3	s11-e4-a4-c1-a1-e1-s1	s11-e4-a5-c4-a2-e1-s1	s11-e4-a6-c7-a3-e1-s1
s5->s38, 0.14	s5-e2-a1-c1-a13-e13-s38	s5-e2-a2-c4-a14-e13-s38	s5-e2-a3-c7-a15-e13-s38

s29->s8, 0.74	s29-e10-a10-c1-a1-e3-s8	s29-e10-a11-c4-a2-e3-s8	s29-e10-a12-c7-a3-e3-s8
s4->s51, 0.98	s4-e2-a1-c1-a16-e17-s51	s4-e2-a2-c4-a17-e17-s51	s4-e2-a3-c7-a18-e17-s51
s9->s21, 0.13	s9-e3-a1-c1-a7-e7-s21	s9-e3-a2-c4-a8-e7-s21	s9-e3-a3-c7-a9-e7-s21
s10->s2, 0.19	s10-e4-a4-c1-a1-e1-s2	s10-e4-a5-c4-a2-e1-s2	s10-e4-a6-c7-a3-e1-s2
s6->s21, 0.12	s6-e2-a1-c1-a7-e7-s21	s6-e2-a2-c4-a8-e7-s21	s6-e2-a3-c7-a9-e7-s21

Table 7: working and backup paths of affected traffic requests in the first scenario with double failure

All of the elements from the working, first and second backup paths are active in the results obtained from the minimum active topology with double failure path protection and therefore our system is protected with that minimum topology.

If we choose any two links be it two aggregation-edge links that are path of both working and backup paths of at least one flow or one core-aggregation and one aggregation-edge links and analyze the results we find that just as in the demonstrated case all the elements are active in the output given by the algorithm.

B) SECOND SCENARIO: DOUBLE FAILURE PROTECTION WITH MIDDLE TRAFFIC

INPUT PARAMETERS

The value of the input parameters for the second scenario is as follows:

- fat-tree degree (k): 6.
- Link capacity (c): 10.
- Switch power consumption (P): 146.
- Port power consumption (pport): 0.9.
- Number of traffic requests: 25.
- Type of traffic: Middle traffic.
- Traffic matrix: The traffic matrix has been randomly generated. The requested bandwidth is in the range [0,1]. We will not get into detail for the traffic matrix in this scenario.

OUTPUT

- Active core switches: c1
- Active aggregation switches: a1, a2, a3, a4, a5,a6,a7, a8, a9, a10, a11, a12, a13, a14, a15, a16, a17, a18
- Active edge switches: e1, e2, e3, e4, e5, e6, e7, e8, e9, e10, e11, e12, e13, e14, e15, e16, e17, e18
- Active links: c1-a1, c1-a4, c1-a7, c1-a10, c1-a13, c1-a16, a1-e1, a1-e2, a1-e3, a4-e4, a4-e5, a4-e6, a7-e7, a7-e8, a7-e9, a10-e10, a10-e11, a10-e12, a13-e13, a13-e14, a13-e15, a16-e16, a16-e17, a16-e18, a2-e2, a2-e3, a5-e4, a5-e5, a5-e6, a8-e7, a8-e8, a8-e9, a11-e10, a11- e11, a11-e12, a14-e13, a14-e14, a14-e15, a17-e16, a17-e17, a17-e18, a3-e2, a3-e3, a6-e4, a6-e5, a6-e6, a9-e7, a9-e8, a9-e9, a12-e10, a12-e11, a12-e12, a15-e13, a15-e14, a15-e15, a18-e16, a18-e17, a18-e18
- Number of active switches: 37
- Number of active ports: 116
- Power consumption: 5506.4
- Utilisation factor: 4.178%

PROTECTION AGAINST DOUBLE FAILURE

In this scenario the only relevant failures would be two failures in aggregation-edge links where one of the links is part of the working path and the other of the backup path of at least one traffic request. If we analyze the output obtained from the algorithm we can see that we have activated all required elements to allocate a working and two backup paths per traffic request.

C) THIRD SCENARIO: DOUBLE FAILURE PROTECTION WITH MIXED TRAFFIC.

INPUT PARAMETERS

The value of the input parameters for the second scenario is as follows:

- fat-tree degree (k): 6.
- Link capacity (c): 10.
- Switch power consumption (P): 146.
- Port power consumption (pport): 0.9.
- Number of traffic requests: 25.
- Type of traffic: Mixed traffic.
- Traffic matrix: The traffic matrix has been randomly generated. The requested bandwidth is in the range [0,1]. We will not get into detail for the traffic matrix in this scenario.

OUTPUT

- Active core switches: c1, c4, c7
- Active aggregation switches: a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15, a16, a17,a18
- Active edge switches: e1, e2, e3, e4, e5, e6, e7, e8, e9, e10, e11, e12, e13, e14, e15, e16, e17, e18
- Active links: c1-a1, c1-a4, c1-a7, c1-a10, c1-a13, c1-a16, c4-a2, c4-a8, c4-a11, c4-a14, c4-a17, c7-a3, c7-a9, c7-a12, c7-a15, a1-e1, a1-e2, a1-e3, a4-e4, a4-e5, a4-e6, a7-e7, a7-e8, a7-e9, a10-e10, a10-e11, a10-e12, a13-e13, a13-e14, a13-e15, a16-e16, a16-e17, a16-e18, a2-e1, a2-e2, a2-e3, a5-e4, a5-e5, a5-e6, a8-e7, a8-e8, a8-e9, a11-e10, a11-e11, a11-e12, a14-e13, a14-e14, a14-e15, a17-e16, a17-e17, a17-e18, a3-e1, a3-e2, a3-e3, a9-e7, a9-e8, a9-e9, a12-e10, a12-e11, a12-e12, a15-e13, a15-e14, a15-e15, a6-e4, a6-e5, a6-e6, a18-e16, a18-e17, a18,e18
- Number of active switches: 39
- Number of active ports: 144
- Power consumption: 5823.6
- Utilisation factor: 4.907%

PROTECTION AGAINST DOUBLE FAILURE

mixed traffic is affected by the same failure scenarios that far traffic did since there are some far traffic requests. Therefore, we can analyze the results obtained from the algorithm to ensure that we have all active elements that will constitute a working and two backup paths for every flow seeing that in any of the double failure scenarios we will have enough active paths to ensure all communications.

5.2 EVALUATION OF ENERGY CONSUMPTION

Adding path protection to the elastic-tree topology has the obvious advantage that the network will be survivable in case of any single or double failure. However this will mean a decrease in the power saving that the traditional elastic-tree had compared to a fat-tree data center. This section will study the results obtained from the algorithms developed to integrate path protection to the elastic-tree in order to asses how much energy is still being saved. to do so we compare the power consumption against the conventional fat-tree data center and the elastic-tree without path protection.
All the results have been obtained from the algorithms “minimum active topology with single failure path protection” and “minimum active topology with double path

protection", implemented on Scilab. The results for the elastic-tree have been obtained from an implementation of the TAH algorithm, also in Scilab while the power consumption for a fat-tree data center has been calculated theoretically.

As introduced earlier in this paper there are several parameters that we obtain from the algorithms that are used to evaluate the energy consumption: number of active switches, number of active ports, power consumed by all active elements and utilisation factor.
Being λ_{ij} is the total requested bandwidth from server i to server j and #servers in the number of server in the data center. $\#servers\ x\ r$ is the total bandwidth capacity of all links between edge switches and servers. Given that each traffic request between servers i and j needs to go through two edge-server links we deduce that the maximum total acceptable load in the data center is $0.5 \#servers\ x\ r$. From here we can define the utilisation index as the load of the data center, being the ratio between the total requested bandwidth and the maximum load of the data center. Calculated by:

$$u = \frac{2 \times \sum_{ij} \lambda_{ij}}{\#servers \times r}$$

For a network without path protection a utilisation index of 100% indicates that the data center is fully loaded. Adding path protection will require for each flow to have more than one path which will result in a decrease of the utilisation factor. For single failure path protection we will need two path per flow resulting in an utilisation index of 50% for a fully loaded data center. In the case of double failure path protection three paths are required and therefore the utilisation index of a fully loaded server would be of around 33.3%.

To calculate the power consumption of a fat-tree we follow the model presented in section 3.5 of this paper. Being P the fixed part of the power consumption of a switch and p the power consumption of a port. It can easily be proven that a k-ary fat-tree data center has $5k^2/4$ switches, $5k^3/4$ ports and $3k^3/4$ links. Being all of them active in a fat-tree, the power consumption will be calculated as:

$$P_{Fat} = \frac{5}{4}k^2P + \frac{5}{4}k^3p$$

The power consumption of an elastic tree topology is dependent on its active switches. All edge switches are active by default to interconnect all servers in the data center, having $k^2/2$ edge switches. If we denote the number of active aggregation switches as x ($x \leq k^2/2$), the number of active core switches y ($y \leq k^2/4$) and the number of active ports as n_p then the total energy consumed by the elastic tree can be calculated as follows:

$$P_{Elastic} = \frac{k^2}{2}P + xP + yP + n_p p$$

Energy saving of the Elastic-tree over the Fat-tree can be defined as:

$$1-\frac{P_{Elastic}}{P_{Fat}}$$

And equally the energy saving of the protected elastic-tree with single or double failure protection can be defined by:

$$1-\frac{P_{ElasticSingle}}{P_{Fat}} \text{ and } 1-\frac{P_{ElasticDouble}}{P_{Fat}}$$

All simulations follow the conditions below:

- The switches are wired in homogeneous three-layered fat-trees of different degrees k = 4,6.
- All links are bidirectional and have a rate r =10 Gbps
- Traffic requests are generated for middle, far and mixed traffic without including near traffic since we do not protect against failures affecting near traffic.
- All traffic requests have a random requested bandwidth in range [0,1] Gbps
- The power consumed by a switch chassis is set to P = 146 watt.
- The power consumed by a port is set to p = 0.9 watt.
- For every simulation traffic is generated with increasing network utilisation being always under the maximum value (50% for single failure and 33.3% for double failure)

Tables 8 and 9 show the results obtained with k = 4 and k = 6 respectively.

ENERGY SAVING WITH MIDDLE TRAFFIC

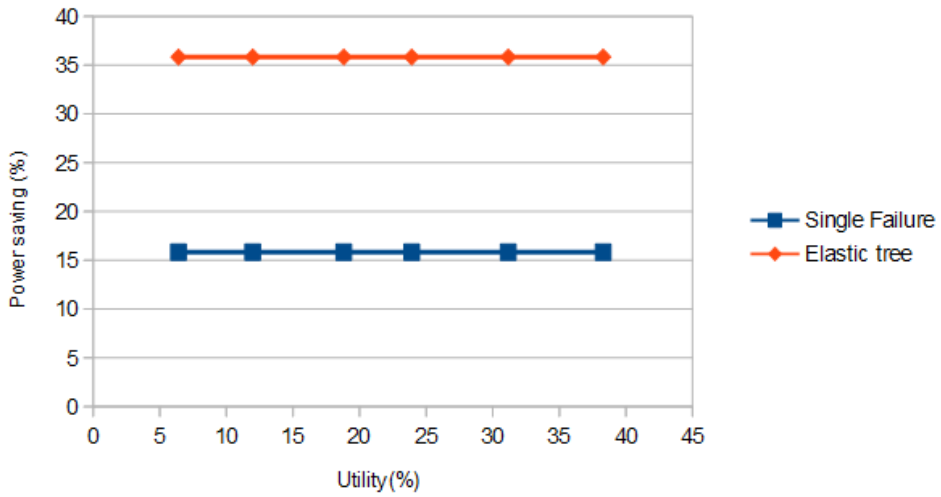

Figure 18: Power saving levels with Far traffic and k=4

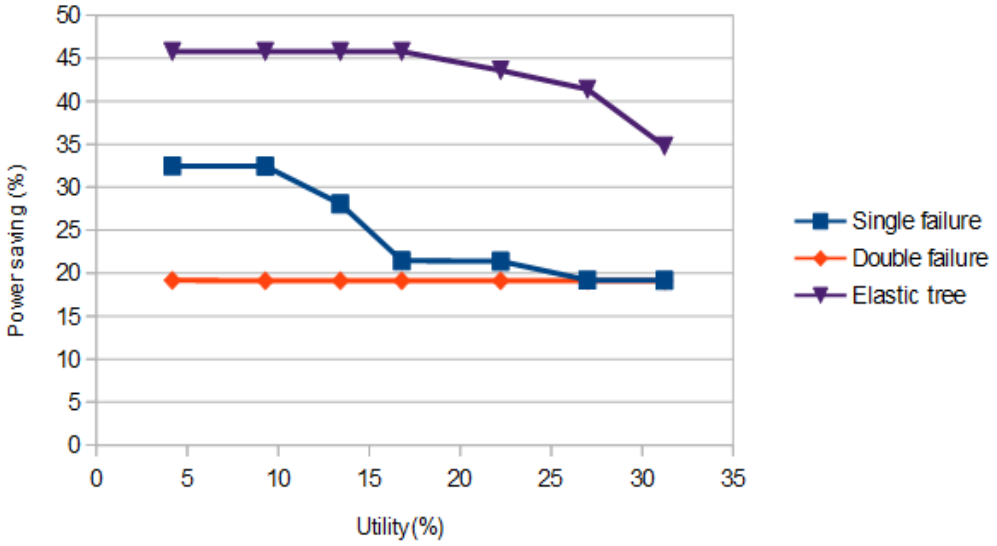

Figure 19: Power saving level with middle traffic and k=4

For middle traffic the core switches and their links to aggregation ones are not involved since both destination and source servers are in the same pod. Therefore, the backup paths does not need to involve any nodes above the aggregation layer resulting in a less pronounced degradation in the power savings compared to the traditional elastic tree that with far traffic. As we can see from figures 18 and 19 double failure does not decrease the energy savings significantly compared to single failure protection in this case which makes it very convenient since we are winning more than we lose.
It should also be noticed that for low loads in the data center many backup paths can profit from the already active switches and ports used as working paths resulting in more savings. Under a very high load we are still able to save 19.17% for both single and double failure.

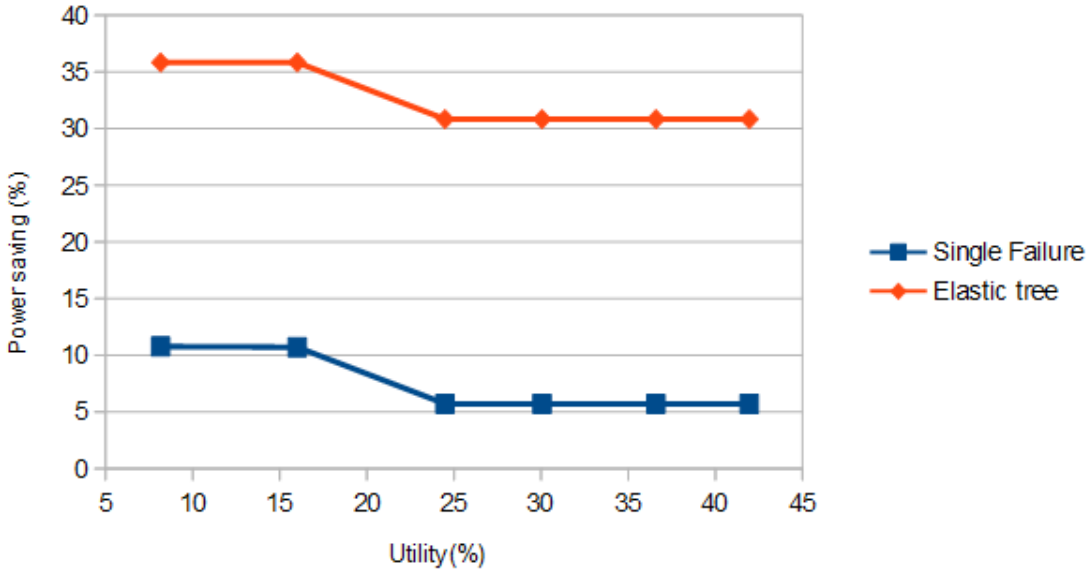

Figure 20: Power saving level with far traffic and k=4

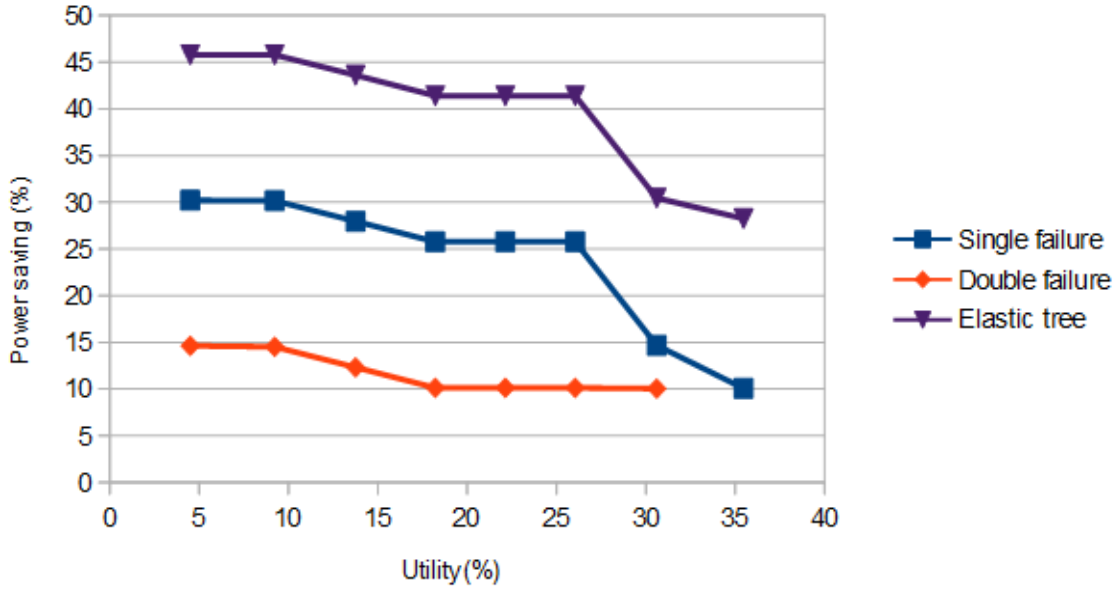

Figure 21: Power saving level with far traffic and k=6

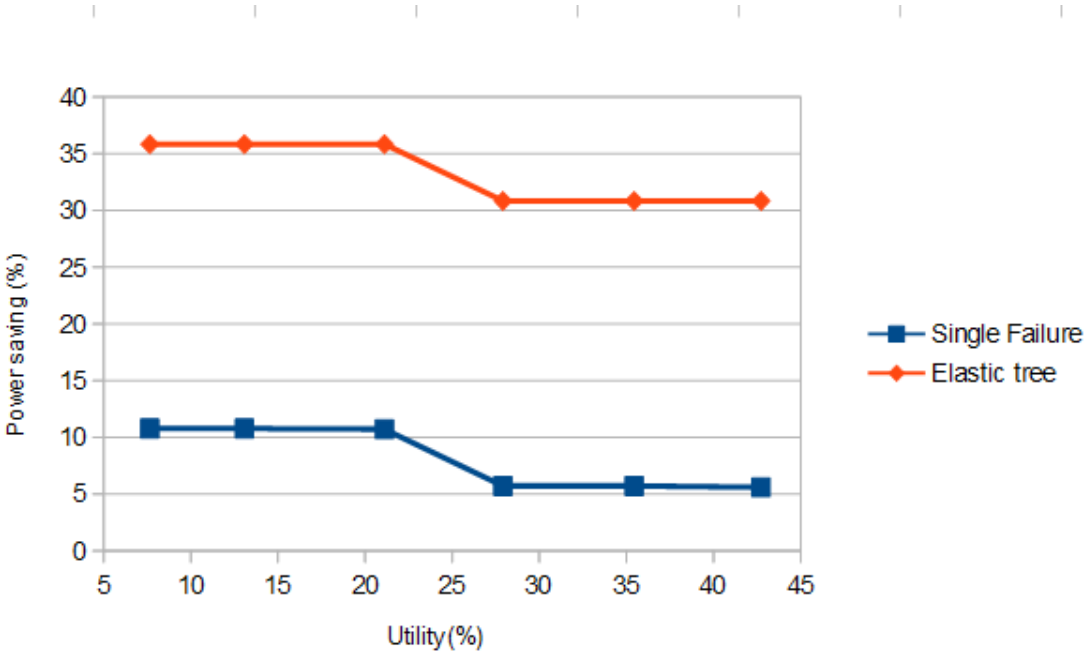

Figure 22: Power saving level with mixed traffic and k=4

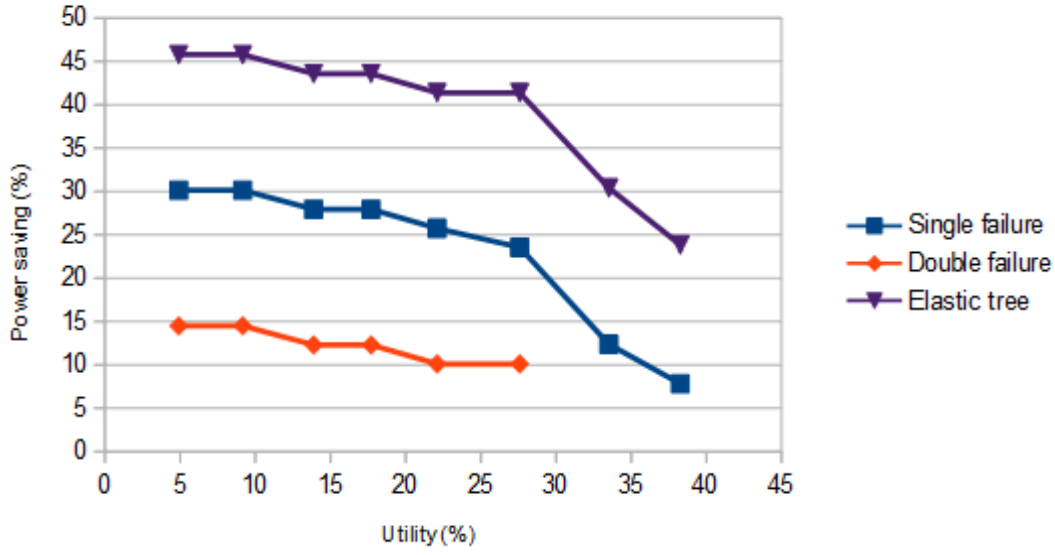

Figure 23: Power saving level with mixed traffic and k=6

Mixed and Far traffic imply a greater power consumption in any case and consequently the protection models also consume a significant amount of energy compared to the conventional Elastic tree. Far traffic has a greater impact than middle traffic due to the involvement of more elements in the network.

For k=4, the savings using single path protection are very small, for small loads we can save around 10% but increasing the load the savings sink as low as 5% compared to the fat-tree.
Better results can be obtained with k=6. For single failure path protection we save around 25-30% for utilization under 30%, after that the savings decrease but proportionally to the savings using traditional elastic-tree topologies. In the case of double failure the savings are significantly inferior. We save between 10-20% but with increased load both single failure and double failure save a similar amount of energy which would mean that for heavy loaded data centers the use of double failure would be justified.

			Single failure				Double Failure				Elastic tree			
Number of requests	Network utility (%)	P_{Fat}	Nb. active switches	Nb. active ports	Consummed power	Power saving (%)	Nb. active switches	Nb. active ports	Consummed power	Power saving (%)	Nb. active switches	Nb. active ports	Consummed power	Power saving (%)
Far traffic														
10	8.1625	2992	18	46	2669.4	10.7821	-	-	-	-	13	24	1919.6	35.8422
20	16	2992	18	48	2671.2	10.7219	-	-	-	-	13	24	1919.6	35.8422
30	24.4875	2992	19	52	2820.8	5.7219	-	-	-	-	14	28	2069.2	30.8422
40	30.05	2992	19	52	2820.8	5.7219	-	-	-	-	14	28	2069.2	30.8422
50	36.5875	2992	19	52	2820.8	5.7219	-	-	-	-	14	28	2069.2	30.8422
60	41.95	2992	19	52	2820.8	5.7219	-	-	-	-	14	28	2069.2	30.8422
Middle traffic														
10	6.4	2992	17	40	2518	15.8422	-	-	-	-	13	24	1919.6	35.8422
20	11.975	2992	17	40	2518	15.8422	-	-	-	-	13	24	1919.6	35.8422
30	18.8125	2992	17	40	2518	15.8422	-	-	-	-	13	24	1919.6	35.8422
40	23.9125	2992	17	40	2518	15.8422	-	-	-	-	13	24	1919.6	35.8422
50	31.1625	2992	17	40	2518	15.8422	-	-	-	-	13	24	1919.6	35.8422
60	38.2875	2992	17	40	2518	15.8422	-	-	-	-	13	24	1919.6	35.8422
Mixed traffic														
10	7.65	2992	18	46	2669.4	10.7821	-	-	-	-	13	24	1919.6	35.8422
20	13.075	2992	18	46	2669.4	10.7821	-	-	-	-	13	24	1919.6	35.8422
30	21.1125	2992	18	48	2671.2	10.7219	-	-	-	-	13	24	1919.6	35.8422
40	27.9125	2992	19	52	2820.8	5.7219	-	-	-	-	14	28	2069.2	30.8422
50	35.45	2992	19	52	2820.8	5.7219	-	-	-	-	14	28	2069.2	30.8422
60	42.725	2992	19	56	2824.4	5.6016	-	-	-	-	14	28	2069.2	30.8422

Table 8: Results obtained with k=4

			Single failure				Double Failure				Elastic tree			
Number of requests	Network utility (%)	P_{Fat}	Nb. active switches	Nb. active ports	Consummed power	Power saving (%)	Nb. active switches	Nb. active ports	Consummed power	Power saving (%)	Nb. active switches	Nb. active ports	Consummed power	Power saving (%)
Far traffic														
25	4.5037	6813	32	92	4754.8	30.2099	39	136	5816.4	14.6279	25	48	3639.2	45.7918
50	9.2222	6813	32	96	4758.4	30.1571	39	144	5823.6	14.5222	25	48	3693.2	45.7918
75	13.7518	6813	33	100	4908	27.9612	40	148	5973.2	12.3264	26	52	3842.8	43.596
100	18.2148	6813	34	104	5057.6	25.7654	41	152	6122.8	10.1306	27	56	3992.4	41.4002
125	22.1333	6813	34	104	5057.6	25.7654	41	152	6122.8	10.1306	27	56	3992.4	41.4002
150	26.0444	6813	34	104	5057.6	25.7654	41	152	6122.8	10.1306	27	56	3992.4	30.447
175	30.6037	6813	39	132	5812.8	14.6807	41	158	6128.2	10.0517	32	74	4738.6	28.2518
Middle traffic														
25	4.1777	6813	31	82	4599.8	32.4849	37	116	5506.4	19.178	25	48	3693.2	45.7918
50	9.2814	6813	31	84	4601.6	32.4585	37	120	5510	19.1252	25	48	3693.2	45.7918
75	13.4111	6813	33	92	4900.8	28.0669	37	120	5510	19.1252	25	48	3693.2	45.7918
100	16.7851	6813	36	106	5351.4	21.4531	37	120	5510	19.1252	25	48	3693.2	45.7918
125	22.2185	6813	36	110	5355	21.4002	37	120	5510	19.1252	26	52	3842.8	43.596
150	26.9889	6813	37	114	5504.6	19.2044	37	120	5510	19.1252	27	56	3882.4	41.4002
175	31.2259	6813	37	116	5506.4	19.1780	37	120	5510	19.1252	30	70	4443	34.7864
Mixed traffic														
25	4.9074	6813	32	96	4758.4	30.1571	39	144	5823.6	14.5222	25	48	3693.2	45.7918
50	9.1592	6813	32	96	4758.4	30.1571	39	144	5823.6	14.5222	25	48	3693.2	45.7918
75	13.8740	6813	33	100	4908	27.9612	40	148	5973.2	12.3264	26	52	3842.8	43.5960
100	17.6962	6813	33	100	4908	27.9612	40	148	5973.2	12.3264	26	52	3842.8	43.5960
125	22.0925	6813	34	104	5057.6	25.7654	41	152	6122.8	10.1306	27	56	3992.4	41.4002
150	27.5852	6813	35	108	5207.2	23.5696	41	152	6122.8	10.1306	27	56	3992.4	41.4002

CONCLUSIONS

The Fat-tree topology has been conceived as a topology for data centers characterized by its low oversubscription and high availability while elastic-tree has been proposed to reduce the energy consumed by a fat-tree data center deactivating a series of links and switches that are not necessary to route the traffic of the data center. Elastic-tree saves a significant amount of energy but does so at expense of the availability of the system.

In this paper, a protected version ef Elastic-tree has been proposed to ensure survivability of the network in case of single and double failure adding one backup path for single failure protection and two for double failure. The simulation results show that the protected elastic tree topology saves a considerable amount of energy compared to a fat-tree topology although it consumes more than a conventional elastic-tree. With higher load in the server the power saving of single failure protected elastic tree and double failure protected elastic tree converge being the double failure more beneficial allowing to protect from any scenario of double failure while still saving energy. Protected Elastic-tree in both of their versions is more efficient with middle traffic than far and mixed traffic since less elements of the network are involved.

REFERENCES

[1]https://www.nrdc.org/resources/americas-data-centers-consuming-and-wasting-growing-amounts-energy

[2] Worldwide electricity used in data centers. IOP science

[3] Data center assessment
https://www.nrdc.org/sites/default/files/data-center-efficiency-assessment-IB.pdf

[4] C.Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing.

[5] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, N. McKeown. ElasticTree: Saving Energy in Data Center Networks

[6] A. Greenberg, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network.

[7] R. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric.

[8] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. DCell: A Scalable and Fault-Tolerant Network Structure for Data Centers.

[9] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers.

[10] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture.

[11] M. Arregoces, M. Portoloni. Data center fundamentals

[12] https://www.paloaltonetworks.com/documentation/glossary/what-is-a-data-center

[13] M. Rouse, Data availability.
http://searchstorage.techtarget.com/definition/data-availability

[14] The linux information project http://www.linfo.org/scalable.html

[15] M. Rouse Security http://searchsecurity.techtarget.com/definition/security

[16] https://en.wikipedia.org/wiki/Computer_performance

[17] What Is Manageability? http://www.ni.com/white-paper/14415/en/

[18] C. E. Leiserson. Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing

[19] P. Mahadevan, P. Sharma, S. Banerjee and P. Ranganathan. A power benchmarking framework for network devices.

[20] Simon Billemont. First steps with the JIMS package for Scilab

ANNEX

PSEUDOCODE FOR TAH ALGORITHM

The following section presents the TAH algorithm in pseudocode according to the description found in the previous section.

```
for each p ∈ P
    for each e ∈ Ep
        for each a ∈ Ap
            LEdge_{p,e}^{up} += F(e → a)
            LEdge_{p,e}^{down} += F(a → e)
        LEdge_{p,e}^{up} = LEdge_{p,e}^{up}/r
        LEdge_{p,e}^{down} = LEdge_{p,e}^{down}/r
    for each a ∈ Ap & c ∈ C where c ↔ a
        LAgg_p^{up} += F(a → c)
        LAgg_p^{down} += F(c → a)
    LAgg_p^{up} = LAgg_p^{up}/r
    LAgg_p^{down} = LAgg_p^{down}/r
for each p ∈ P
    for each e ∈ Ep
        if(LEdge_p^{up} > NAgg_p^{up}) NAgg_p^{up} = LEdge_p^{up}
    NAgg_p^{down} =⌈(LAgg_p^{down}/(k/2))⌉
    NAggp = max{NAgg_p^{up}, NAgg_p^{up},1}
    if(LAgg_p^{up} > NCore) NCore = LAgg_p^{up}
NCore =⌈NCore⌉
```

EXPLAINED CODE OF FAT-TREE STRUCTURE IMPLEMENTATION

Most of the code for the fat tree structure is trivial and therefore in this section only an explanation of the method to generate the fat tree is presented. The rest of the code can be found in the appendix.

```
public FatTree(Integer kk,Double cc,Integer Pp,double pportt){
    k = kk;
    c =cc;
    P =Pp;
    pport = pportt;
```

In the first part of the method we assign the values given by the user to their respective parameters.

```
Integer lc = 1;
Integer la = 1;
Integer le = 1;
Integer ls =1;
node aggreg = new node();
ArrayList<node> agN = new ArrayList<node>();
for(Integer i=0;i<k/2;++i){
    String lab = lc.toString();
    Boolean b=false;
    if(i==0)b=true;
    node core = new node(P,pport,b,"c"+lab);
    ++lc;
```

The fat tree consists of $(k/2)^2$ core nodes. They can be divided in k/2 groups inside of which all cores will be identical, connected to the same switches in every pod. We start by creating one core of each group from where we will proceed to create the nodes in lower layers these are connected to. In this piece of code we can observe how we create k/2 cores. Only the first one is active by default.

```
for(Integer j=0;j<k;++j){
    Integer d = ((j*k/2)+i);
    la = 1;
    la = la+d;
    String lab1 = la.toString();
```

Then we move on to generating the nodes in the aggregation layer and adding them as children to the corresponding node. We generate k aggregation nodes per core, each one of them in a different pod in the network.

All aggregation nodes in a pod are connected to all edge switches, and only the one connected to the core node is active by default to ensure all servers are accessible from anywhere in the network. That is why we differentiate between aggregation switches when generating them:

```
if(i==0){
    aggreg = new node(P,pport,true,"a"+lab1);
    core.addchild(aggreg,c,true);
        for(Integer z=0;z<k/2;++z){
            String lab2 = le.toString();
            node edge = new node(P,pport,true,"e"+lab2);
            aggreg.addchild(edge,c,true);
            ++le;
            for(Integer x=0;x<k/2;++x){
                String lab3 = ls.toString();
                node server = new node(0,0,true,"s"+lab3);
                edge.addchild(server,c,true);
                ++ls;
            }
        }
        agN.add(aggreg);
}
```

If the aggregation switch is connected to the first core, we generate the aggregation switch indicating it will be active and we add it as the core's child. Then we move to generate the edge switches inside the same pod as the aggregation switch (k/2). all edge switches will be active and will have k/2 servers connected to it that will be the leaves of the tree.

```
else{
    node agg = new node(P,pport,false,"a"+lab1);
    agg.copy(agN.get(j));
    agg.deactivateAll();
    agg.setLabel("a"+lab1);
    agg.setAct(false);
    core.addchild(agg, c, false);

}
```

If on the contrary the aggregation switch is not one of the first node's children, we create an inactive node and instead of generating edge nodes and servers we copy the connections from the node in the pod that is connected to the first core and from where we have generated the lower layers of the network.

```
cores.add(core);
for(int z=1;z<k/2;++z){
    lab = lc.toString();
    node core2 = new node();
    core2.copy(core);
    core2.deactivateAll();
    core2.setLabel("c"+lab);
    core2.setAct(false);
    ++lc;
    cores.add(core2);
}
```

Finally we generate the rest of the cores in every group as inactive copies of the core node that we were generating.

Single failure:

EXPLAINED CODE OF MINIMUM ACTIVE TOPOLOGY WITH SINGLE FAILURE PATH PROTECTION ALGORITHM

In this section the code for the single failure can be found along with an explanation when deemed necessary.

Main function:

```
javaclasspath("C:\Users\Elena\Documents\NetBeansProjects\FatTree\dist\FatTree.jar");
jimport('fattree.FatTree');
jimport('fattree.node');
```

We first import our java code to be able to use the fatTree and node classes implemented in java as previously described.

```
fd=mopen('C:\Users\Elena\Desktop\ElasticTree\Input.txt','r');
x=mgetl(fd);
XX=strsplit(x,ascii(9));
k=int32(strtod(XX(1)));
c=strtod(XX(2));
p=int32(strtod(XX(3)));
pport=strtod(XX(4));
n=int32(strtod(XX(5)));
t=int32(strtod(XX(6)));
tree = FatTree.new(k,c,p,pport);
s=string(0);
re=struct('core',string,'aggreg',string,'edge',string,'links',string);
j=7;
mat = zeros((k^2)/2,(k^2)/2);
lam=0;
for i=1:n;
        s=strtod(XX(j));
        j=j+1;
        d=strtod(XX(j));
        j=j+1;
        b=strtod(XX(j));
        lam=lam+b;
        j=j+1;
        sd=double(k/2);
        es=ceil(s/sd);
        ed =ceil(d/sd);
        mat(es,ed)=mat(es,ed)+b;
end
```

The input can be found in the document Input.txt in this case, we open and read it extracting all the parameters that we need for our algorithm (k,c,p,pport,n,t) n and t are new parameters that have not been previously described, they are auxiliary parameters. n indicates the amount of traffic requests to be considered and t indicates the type of traffic, it can be far, middle or mixed. From the n traffic requests we generate a matrix mat. The original input contains the traffic requests between servers, but the matrix we generate aggregates these requests to only consider them at edge layer level.

```
ptt=list();
for i=1:(k^2)/2
    for j=1:(k^2)/2
        if (mat(i,j)~=0) then
            s='e'+string(i);
            d='e'+string(j);
            b=mat(i,j);
            while(b>int32(c))
            b=b-int32(c);
                f=findpath(s,d,int32(c),tree);
                ptt($+1)=f.path;
            end
            f=findpath(s,d,b,tree);
            ptt($+1)=f.path;
            if(f.capacity==-1) then
                disp('The traffic request couldnt be acommodated');
            end
        end
    end
end
```

Once we have all the necessary parameters we start by allocating the working paths. we divide the requests in case that the bandwidth is higher to the capacity of a link since it would be impossible to allocate. Therefore we can have several working paths between to same edge servers if the number of requests is very high. All working paths are saved in a list to ensure while finding the backup path that they will be completely disjoint.

```
x=1;
ptx=1;
for i=1:(k^2)/2
    for j=1:(k^2)/2
        if (mat(i,j)~=0) then
            s='e'+string(i);
            d='e'+string(j);
            b=mat(i,j);
            while(b>int32(c))
                b=b-int32(c);
                g=findBackPath(s,d,int32(c),tree,ptt(ptx));
                ptx=ptx+1;
            end
            g=findBackPath(s,d,b,tree,ptt(ptx));
                ptx=ptx+1;
        end
    end
end
```

After allocating all working paths, backup paths for all flows have to be found. We do so in a similar manner to the allocation of working paths with the only difference that we use the working path to ensure that the same aggregation switches are not included in the backup path.

```
re.core=jinvoke(tree,"getCore");
re.aggreg=jinvoke(tree,"getAgg");
re.edge=jinvoke(tree,"getEdg");
re.links=jinvoke(tree,"getLinks");
wri=re.core+ascii(10)+re.aggreg+ascii(10)+re.edge+ascii(10)+re.links;
mclose('C:\Users\Elena\Desktop\ElasticTree\Input.txt');
fd=mopen('C:\Users\Elena\Desktop\ElasticTree\Output.txt','a');
```

Once we allocated all paths we generate the output that can be found in Output.txt in our case. The output will consist on a list of all switches and links that need to be active to rute all traffic requests.

```
if t==1 then
    fd2=mopen('C:\Users\Elena\Desktop\ElasticTree\StatisticMx.txt','a');
end
if t==2 then
    fd2=mopen('C:\Users\Elena\Desktop\ElasticTree\StatisticM.txt','a');
end
if t==3 then
    fd2=mopen('C:\Users\Elena\Desktop\ElasticTree\StatisticF.txt','a');
end
NAcore=jinvoke(tree,'NAcore');
NAagg = jinvoke(tree,'NAagg');
NAEd=jinvoke(tree,'NAEd');
NAlink=jinvoke(tree,'NAlink');
Sact=NAcore+NAagg+NAEd;
Pact=2*NAlink;
Power = double(Sact*p)+(double(Pact)*pport);
nser=k*k*k/4;
a1=double(2*lam);
a2=double(c*nser);
Utility=(a1/a2)*100;
stri=string(Sact)+ascii(9)+string(Pact)+ascii(9)+string(Power)+ascii(9)+
string(Utility)+ascii(9)+ascii(9);
mputl(stri,fd2);
mputl(wri,fd);
mclose('C:\Users\Elena\Desktop\ElasticTree\Output.txt');
if t==1 then
    fd2=mclose('C:\Users\Elena\Desktop\ElasticTree\StatisticMx.txt');
end
if t==2 then
    fd2=mclose('C:\Users\Elena\Desktop\ElasticTree\StatisticM.txt');
end
if t==3 then
    fd2=mclose('C:\Users\Elena\Desktop\ElasticTree\StatisticF.txt');
end
```

Finally the statistics are generated. The statistics will be added in a different document depending on the type of traffic that has been analyzed. Each document will contain the number of active switches, of active ports, the power consumed by the network and the utility.

Findpath function:

```
function result=findpath(s,d,b,tree);
    n=jinvoke(tree,"getNCore");
    j=0;
    maxCap=int32(0);
    result=struct('capacity',maxCap,'path',list());
    result.path=list();
    for j=0:n-1
        nod=jinvoke(tree,"getCore",j);
        bol=jinvoke(nod,"getAct");
        if bol then
            pcs = path(nod,s,1,%inf);
            pcd = path(nod,d,1,%inf);
            cap = min(pcs.capacity,pcd.capacity);
            if cap>maxCap then
                maxCap=cap;
                result.path=list();
                pcs.path=reverse(pcs.path);
                result.path = lstcat(pcs.path,pcd.path);
                result.capacity=maxCap;
            end
        end
    end
```

Findpath is the main function in charge of allocation the working flows. We start by browsing all paths connecting to the source and destination edge switch. To do so we find paths to both switches from all active core switches invoking the *path* function. From every path that we find we assess if the capacity is the maximum one. In case that we find it is we take that path as provisional working path until we find one with higher residual bandwidth. When all paths from every active core have been evaluated we should have found a working path.

```
if maxCap<b then
    minPower=%inf;
    rest=struct('power',-1,'capacity',-1,'path',list());
    for k=0:(n-1)
        nod=jinvoke(tree,"getCore",k);
        pcS=actPath(nod,s,1,%inf);
        pcD=actPath(nod,d,1,%inf);
        pow=pcS.power+pcD.power;
        cap=min(pcS.capacity,pcD.capacity);
        if pow<minPower & cap>b then
            minPower=pow;
            pcS.path=reverse(pcS.path);
            result.path=lstcat(pcS.path,pcD.path);
            result.capacity=minPower;
        end
    end
```

It is possible that the working path that we have found does not have enough capacity to accommodate the traffic demand. In this case we browse all paths including those that contain inactive components. We find paths from every core node to both source and

destination edge switches by invoking actPath function. In this case we are looking for the path with the least additional power consumption instead of the one with maximum capacity. Therefore, for every path that we find we check if the power consumption is the minimal found until then. If it is we select the path as provisional working path.
By the time all paths have been inspected we should have a working path.

```
    if size(result.path)==0 then
        disp('all traffic requests could not be accomodated');
    else
        aux=1;
        x=int32((size(result.path)/2)+1);
        if isequal(result.path(x-2),result.path(x+2)) then
            aux=3;
            activatePath(tree,result.path,aux);
        elseif isequal(result.path(x-1),result.path(x+1)) then
            aux=2;
            activatePath(tree,result.path,aux);
            else activatePath(tree,result.path,aux);
        end
    end
end
```

If none of the methods described above find a path that has enough capacity to accommodate the requests this means that the network does not have enough capacity to allocate all traffic and the algorithm will display a message.
If we have found a path that contains inactive elements we activate them invoking the method *activatePath*.

```
    if(size(result.path)~=0) then
    aux=1;
        x=int32((size(result.path)/2)+1);
        if isequal(result.path(x-2),result.path(x+2)) then
            aux=3;
            refreshCap(tree,result.path,b,aux);
        elseif isequal(result.path(x-1),result.path(x+1)) then
            aux=2;
            refreshCap(tree,result.path,b,aux);
            end
         refreshCap(tree,result.path,b,aux);
    end
endfunction
```

Finally we refresh the capacity of the working path invoking the method *refreshCap* to decrease it representing that part of its capacity is dedicated to the traffic flow.

Path function:

```
function pc=path(co,s,j,cap)
    p=list();
    pc = struct('capacity',-1,'path',p);
    pc.path=list();
    res=jinvoke(co,"getLabel");
    if isequal(res,s) then
        pc.path(j)=res;
        pc.capacity=cap;
    elseif jinvoke(co,"hasChildren") then
        for z=0:(jinvoke(co,"getNChild")-1)
            nod=jinvoke(co,"getChild",z);
            bol=jinvoke(nod,"getAct");
            if bol then
                aux=path(nod,s,j+1,jinvoke(co,"getCapacity",z));
                if ~isequal(aux.capacity,-1) then
                    pc.path(j)=jinvoke(co,"getLabel");
                    for m=j+1:size(aux.path)
                        pc.path(m)=aux.path(m);
                    end
                    pc.capacity=min(aux.capacity,cap);
                end
                jremove(res);
            end
        end
    end
```

Path is a recursive function that searches for a path between a given core node and the edge switch with label s. The function ends when the base case is met, when we find the edge switch s. Otherwise we recursively invoke the function *path* exploring all active children of the core node co and its children. Besides the path connecting the core and edge node, the path function also gives the capacity of the path, corresponding to the minimum capacity of a link in the path.

ActPath function:

```
function pc=actPath(co,s,j,cap)
    pc = struct('power',-1,'capacity',-1,'path',list());
    pc.path=list();
    res=jinvoke(co,"getLabel");
    if isequal(res,s) then
        pc.path(j)=res;
        pc.capacity=cap;
        pc.power=jinvoke(co,"getP")+(jinvoke(co,"getNact")
        *jinvoke(co,"getPport"));
    elseif jinvoke(co,"hasChildren") then
        for z=0:(jinvoke(co,"getNChild")-1)
            nod=jinvoke(co,"getChild",z);
                cap=jinvoke(co,"getCapacity",z);
                aux=actPath(nod,s,j+1,cap);
                if ~isequal(aux.capacity,-1) then
                    pc.path(j)=jinvoke(co,"getLabel");
                    for m=j+1:size(aux.path)
                        pc.path(m)=aux.path(m);
                    end
                    if jinvoke(co,"getAct") then
                    pow=jinvoke(co,"getP")+((jinvoke(co,"getNact")+1)
                    *jinvoke(co,"getPport"));
                else
                pow=jinvoke(co,"getP")+(jinvoke(co,"getNact")
                *jinvoke(co,"getPport"));
                end
                    pc.power=aux.power+pow;
                    pc.capacity=min(aux.capacity,cap);
                end
                jremove(res);
        end
    end
endfunction
```

ActPath is a recursive function that works in a similar manner to the *path* function. It ends when the base case is met, this happens when we have found the edge switch with label s. There are a few differences between *path* and *actPath* functions. *actPath* assists in finding a path with minimum additional power consumption and, therefore, besides returning the path and its capacity it also return the value of the power consumed by the elements in the path. another difference is that not only active switches are explored, *actPath* searches for a path regardless of the elements it includes.

Reverse function:

```
function r=reverse(pcs)
    r=list();
    m=size(pcs)-1;
    for n=2:size(pcs)
        r(m)=pcs(n);
        m=m-1;
    end
endfunction
```

reverse function is a very simple auxiliary method. Given that we find working and backup paths in two parts, from source to core and the from core to destination and both are calculated through the same function we later change the format of one of the paths removing the core node and reversing the elements in it so it follows a logical order when combining both paths.

RefreshCap function:

```
function []=refreshCap(t,path,b,i)
    x=int32((size(path)/2)+1);
    nod=jinvoke(t,"findCore",path(x));
    if i==1 then
        for m=1:size(path)/2
            nod=jinvoke(t,"findNode",path(m+1));
            jinvoke(nod,"changeCap",path(m),b);
        end
        for m=(size(path)/2)+1:size(path)-1
            nod=jinvoke(t,"findNode",path(m));
            jinvoke(nod,"changeCap",path(m+1),b);
        end
    end
    if i==2 then
        for m=1:(size(path)/2)-1
            nod=jinvoke(t,"findNode",path(2));
            jinvoke(nod,"changeCap",path(1),b);
        end
            nod=jinvoke(t,"findNode",path(4));
            jinvoke(nod,"changeCap",path(5),b);
    end
endfunction
```

RefreshCap is a function whose mission is to update the capacity of the working path indicating that it accommodates a flow. All the paths that the algorithm finds follow the same format: source edge switch - source aggregation switch - core switch - destination aggregation switch - destination edge switch. This type of path is only needed for far traffic, with middle traffic the aggregation switches are the same and we don't need to go all the way up to the core. Therefore, *refreshCap* differentiates the type of traffic and in case of far traffic updates all link capacities in the path while in case of middle traffic only updates capacities up to aggregation layer.

ActivatePath function:

```
function []=activatePath(tree,path,i)
    x=int32((size(path)/2)+1);
    nod=jinvoke(tree,"findCore",path(x));
    if i==1 then
        for m=x:size(path)-1
            nod=jinvoke(tree,'findNode',path(m));
            jinvoke(nod,'setAct',%T);
            jinvoke(nod,'activate',path(m+1));
        end
        m=m+1;
        jinvoke(tree,'activate',path(m));
        for m=x:-1:2
            nod=jinvoke(tree,'findNode',path(m));
            jinvoke(nod,'setAct',%T);
            jinvoke(nod,'activate',path(m-1));
        end
        m=m-1;
            jinvoke(tree,'activate',path(m));
    end
    if i==2 then
        for m=x+1:size(path)-1
            nod=jinvoke(tree,'findNode',path(m));
            jinvoke(nod,'setAct',%T);
            jinvoke(nod,'activate',path(m+1));
        end
        m=m+1;
        jinvoke(tree,'activate',path(m));
        for m=x-1:-1:2
            nod=jinvoke(tree,'findNode',path(m));
            jinvoke(nod,'setAct',%T);
            jinvoke(nod,'activate',path(m-1));
        end
        m=m-1;
            jinvoke(tree,'activate',path(m));
    end
```

ActivatePath is the function in charge of activating all elements in a working or backup path that have not been previously activated. As with the *refreshCap* function we have to differentiate between types of traffic to make sure we only activate the relevant elements in the path. To activate elements we use the methods defined in the *fatTree* and node classes to activate both switches and links.

findBackPath function:

```
function result=findBackPath(s,d,b,tree,pt);
    n=jinvoke(tree,"getNCore");
    j=0;
    maxCap=int32(0);
        result=struct('capacity',maxCap,'path',list());
                result.path=list();
    for j=0:n-1
        nod=jinvoke(tree,"getCore",j);
        bol=jinvoke(nod,"getAct");
        if bol then
            ag1=int32(size(pt)/2);
            ag1=pt(ag1);
            ag2=int32(size(pt)/2+2);
            ag2=pt(ag2);
            pcs = Bpath(nod,s,1,%inf,ag1);
            pcd = Bpath(nod,d,1,%inf,ag2);
            cap = min(pcs.capacity,pcd.capacity);
            if cap>maxCap then
                maxCap=cap;
                result.path=list();
                pcs.path=reverse(pcs.path);
                result.path = lstcat(pcs.path,pcd.path);
                result.capacity=maxCap;
            end
        end
    end
```

findBackPath is the equivalent to *findpath* but to allocate the backup path of a flow.
We start by trying to find a backup path from every active core to the source and destination edge switches separately invoking the method *Bpath*. One of the parameters required by the *Bpath* function is the aggregation switch that should be avoided in the backup path. The path from the core to the source edge switch will exclude the source aggregation switch of the work path while the path between the core and the destination edge switch will do the same with the destination aggregation switch from the working path. As in *findpath* we try to find a suitable path with the maximum available capacity.

```
    if maxCap<b then
        minPower=%inf;
        rest=struct('power',-1,'capacity',-1,'path',list());
        for k=0:(n-1)
            nod=jinvoke(tree,"getCore",k);
            pcS=actBPath(nod,s,1,%inf,ag1);
            pcD=actBPath(nod,d,1,%inf,ag2);
            pow=pcS.power+pcD.power;
            cap=min(pcS.capacity,pcD.capacity);
            if pow<minPower & cap>b then
                minPower=pow;
                pcS.path=reverse(pcS.path);
                result.path=lstcat(pcS.path,pcD.path);
                result.capacity=minPower;
            end
        end

    end
```

Yet again, if we are unable to find a path with sufficient capacity available we move on to find a path that may contain inactive elements still excluding the aggregation switches from the working path. Seeking to find the one with minimum additional power consumption.

```
    if size(result.path)==0 then
            disp('not all transactions are protected');
        else
            aux=1;
            x=int32((size(result.path)/2)+1);
            if isequal(result.path(x-2),result.path(x+2)) then
                aux=3;
            elseif isequal(result.path(x-1),result.path(x+1)) then
                aux=2;
                end
            activatePath(tree,result.path,aux);
        end
endfunction
```

Finally, in case that the backup path that we have found contains inactive elements we invoke the method *activatePath* to activate them. *FindBackPath* will return the backup path and its capacity/power consumption.

Bpath function:

```
function pc=Bpath(co,s,j,cap,ag)
    p=list();
    pc = struct('capacity',-1,'path',p);
    pc.path=list();
    res=jinvoke(co,"getLabel");
    if isequal(res,s) then
        pc.path(j)=res;
        pc.capacity=cap;
    elseif ~isequal(res,ag) & jinvoke(co,"hasChildren") then
        for z=0:(jinvoke(co,"getNChild")-1)
            nod=jinvoke(co,"getChild",z);
            bol=jinvoke(nod,"getAct");
            if bol then
                aux=Bpath(nod,s,j+1,jinvoke(co,"getCapacity",z),ag);
                if ~isequal(aux.capacity,-1) then
                    pc.path(j)=jinvoke(co,"getLabel");
                    for m=j+1:size(aux.path)
                        pc.path(m)=aux.path(m);
                    end
                    pc.capacity=min(aux.capacity,cap);
                end
                jremove(res);
            end
        end
    end
endfunction
```

Bpath function works just like the *path* function but to find a path between the core node “co” and the edge switch “s” that must exclude the the aggregation switch “ag”, if there is one. The function will return the path found or an empty path if there is none.

actBPath function:

```
function pc=actBPath(co,s,j,cap,ag)
    pc = struct('power',-1,'capacity',-1,'path',list());
    pc.path=list();
    res=jinvoke(co,"getLabel");
    if isequal(res,s) then
        pc.path(j)=res;
        pc.capacity=cap;
        pc.power=jinvoke(co,"getP")+(jinvoke(co,"getNact")
        *jinvoke(co,"getPport"));
    elseif jinvoke(co,"hasChildren") & ~isequal(res,ag) then
        for z=0:(jinvoke(co,"getNChild")-1)
            nod=jinvoke(co,"getChild",z);
                cap=jinvoke(co,"getCapacity",z);
                aux=actBPath(nod,s,j+1,cap,ag);
                if ~isequal(aux.capacity,-1) then
                    pc.path(j)=jinvoke(co,"getLabel");
                    for m=j+1:size(aux.path)
                        pc.path(m)=aux.path(m);
                    end
                    if jinvoke(co,"getAct") then
                        pow=jinvoke(co,"getP")+((jinvoke(co,"getNact")+1)
                        *jinvoke(co,"getPport"));
                    else
                        pow=jinvoke(co,"getP")+(jinvoke(co,"getNact")
                        *jinvoke(co,"getPport"));
                    end
                    pc.power=aux.power+pow;
                    pc.capacity=min(aux.capacity,cap);
                end
                jremove(res);
        end
    end
endfunction
```

ActBPath is a recursive function that works similarly to the function *actPath*. *ActBPath* finds a backup path excluding the aggregation switch “ag” that connects the core node “co” and the edge switch “s”. The function returns both the path found and the additional power consumption of the path since we hope to find the one where it is minimum.

EXPLAINED CODE OF MINIMUM ACTIVE TOPOLOGY WITH DOUBLE FAILURE PATH PROTECTION ALGORITHM

As stated in the previous section most of the code has been recycled from the single failure algorithm. Therefore we will only include the new code with explanations where deemed necessary.

Main function:

```
fd=mopen('C:\Users\Elena\Desktop\ElasticTree\Input.txt','r');
x=mgetl(fd);
XX=strsplit(x,ascii(9));
k=int32(strtod(XX(1)));
c=strtod(XX(2));
p=int32(strtod(XX(3)));
pport=strtod(XX(4));
n=int32(strtod(XX(5)));
t=int32(strtod(XX(6)));
tree = FatTree.new(k,c,p,pport);
s=string(0);
re=struct('core',string,'aggreg',string,'edge',string,'links',string);
j=7;
mat = zeros((k^2)/2,(k^2)/2);
lam=0;
for i=1:n;
        s=strtod(XX(j));
        j=j+1;
        d=strtod(XX(j));
        j=j+1;
        b=strtod(XX(j));
        lam=lam+b;
        j=j+1;
        sd=double(k/2);
        es=ceil(s/sd);
        ed =ceil(d/sd);
        mat(es,ed)=mat(es,ed)+b;
end
```

Like for single failure we start by obtaining the input and processing it.

```
ptt=list();
for i=1:(k^2)/2
    for j=1:(k^2)/2
        if (mat(i,j)~=0) then
            s='e'+string(i);
            d='e'+string(j);
            b=mat(i,j);
            while(b>int32(c))
            b=b-int32(c);
                f=findpath(s,d,int32(c),tree);
                ptt($+1)=f.path;
            end
            f=findpath(s,d,b,tree);
            ptt($+1)=f.path;
            if(f.capacity==-1) then
                disp('The traffic request couldnt be acommodated');
            end
        end
    end
end
x=1;
ptx=1;
pttB=list();
for i=1:(k^2)/2
    for j=1:(k^2)/2
        if (mat(i,j)~=0) then
            s='e'+string(i);
            d='e'+string(j);
            b=mat(i,j);
            while(b>int32(c))
                b=b-int32(c);
                g=findBackPath(s,d,int32(c),tree,ptt(ptx));
                ptx=ptx+1;
                pttB($+1)=g.path;
            end
            g=findBackPath(s,d,b,tree,ptt(ptx));
                ptx=ptx+1;
                pttB($+1)=g.path;
        end
    end
end
```

We allocate working and backup paths.

```
ptx=1;
for i=1:(k^2)/2
    for j=1:(k^2)/2
        if (mat(i,j)~=0) then
            s='e'+string(i);
            d='e'+string(j);
            b=mat(i,j);
            while(b>int32(c))
                b=b-int32(c);
                if(size(pttB(ptx))~=0) then
                g2=findBackPath2(s,d,int32(c),tree,ptt(ptx),pttB(ptx));
                ptx=ptx+1;
                end
            end
            if(size(pttB(ptx))~=0) then
            g2=findBackPath2(s,d,b,tree,ptt(ptx),pttB(ptx));
                ptx=ptx+1;
             end
        end
    end
end
```

Once all working and backup paths are allocated we move on to find the second backup path for every flow in the matrix. In this version of the algorithm we have two lists containing the working and backup paths to exclude their aggregation switches ensuring that all three paths are disjoint.

```
re.core=jinvoke(tree,"getCore");
re.aggreg=jinvoke(tree,"getAgg");
re.edge=jinvoke(tree,"getEdg");
re.links=jinvoke(tree,"getLinks");
wri=re.core+ascii(10)+re.aggreg+ascii(10)+re.edge+ascii(10)+re.links;
mclose('C:\Users\Elena\Desktop\ElasticTree\Input.txt');
fd=mopen('C:\Users\Elena\Desktop\ElasticTree\Output.txt','a');
if t==1 then
    fd2=mopen('C:\Users\Elena\Desktop\ElasticTree\StatisticMx2.txt','a');
end
if t==2 then
    fd2=mopen('C:\Users\Elena\Desktop\ElasticTree\StatisticM2.txt','a');
end
if t==3 then
    fd2=mopen('C:\Users\Elena\Desktop\ElasticTree\StatisticF2.txt','a');
end
NAcore=jinvoke(tree,'NAcore');
NAagg = jinvoke(tree,'NAagg');
NAEd=jinvoke(tree,'NAEd');
NAlink=jinvoke(tree,'NAlink');
Sact=NAcore+NAagg+NAEd;
Pact=2*NAlink;
Power = double(Sact*p)+(double(Pact)*pport);
nser=k*k*k/4;
a1=double(2*lam);
a2=double(c*nser);
Utility=(a1/a2)*100;
stri=string(Sact)+ascii(9)+string(Pact)+ascii(9)+string(Power)+ascii(9)
+string(Utility)+ascii(9)+ascii(9);
mputl(stri,fd2);
mputl(wri,fd);
mclose('C:\Users\Elena\Desktop\ElasticTree\Output.txt');
if t==1 then
    fd2=mclose('C:\Users\Elena\Desktop\ElasticTree\StatisticMx2.txt');
end
if t==2 then
    fd2=mclose('C:\Users\Elena\Desktop\ElasticTree\StatisticM2.txt');
end
if t==3 then
    fd2=mclose('C:\Users\Elena\Desktop\ElasticTree\StatisticF2.txt');
end
```

To finish we generate the statistics and output and we write them in the corresponding file.

findBackPath2 function:

```
function result=findBackPath2(s,d,b,tree,pt,ptb);
    n=jinvoke(tree,"getNCore");
    j=0;
    maxCap=int32(0);
        result=struct('capacity',maxCap,'path',list());
                result.path=list();
    for j=0:n-1
        nod=jinvoke(tree,"getCore",j);
        bol=jinvoke(nod,"getAct");
        if bol then
            ag1=int32(size(pt)/2);
            ag1=pt(ag1);
            ag2=int32(size(pt)/2+2);
            ag2=pt(ag2);
            ag3=int32(size(ptb)/2);
            ag3=ptb(ag3);
            ag4=int32(size(ptb)/2+2);
            ag4=pt(ag4);
            pcs = Bpath2(nod,s,1,%inf,ag1,ag3);
            pcd = Bpath2(nod,d,1,%inf,ag2,ag4);
            cap = min(pcs.capacity,pcd.capacity);
            if cap>maxCap then
                maxCap=cap;
                result.path=list();
                pcs.path=reverse(pcs.path);
                result.path = lstcat(pcs.path,pcd.path);
                result.capacity=maxCap;
            end
        end
    end
```

findBackPath2 is the main function in charge of allocating the second backup path. To do so we start by browsing all active paths from each active core in the network to both the source and the destination edge switches through the function *bpath2*. We ought to make sure that the path that we find does not contain any of the aggregation switches found in the working and backup paths. from the paths found the selected one will be the one with the greatest maximum available capacity.

```
    if maxCap<b then
        minPower=%inf;
        rest=struct('power',-1,'capacity',-1,'path',list());
        for k=0:(n-1)
            nod=jinvoke(tree,"getCore",k);
            pcS=actBPath2(nod,s,1,%inf,ag1,ag3);
            pcD=actBPath2(nod,d,1,%inf,ag2,ag4);
            pow=pcS.power+pcD.power;
            cap=min(pcS.capacity,pcD.capacity);
            if pow<minPower & cap>b then
                minPower=pow;
                pcS.path=reverse(pcS.path);
                result.path=lstcat(pcS.path,pcD.path);
                result.capacity=minPower;
            end
        end

    end
    if size(result.path)==0 then
            disp('not all transactions are protected');
        else
            aux=1;
            x=int32((size(result.path)/2)+1);
            if isequal(result.path(x-2),result.path(x+2)) then
                aux=3;
            elseif isequal(result.path(x-1),result.path(x+1)) then
                aux=2;
                end
            activatePath(tree,result.path,aux);
        end
endfunction
```

If we are unable to find a path with enough capacity to allocate the traffic flow we browse all paths in the network connecting every core to the source and destination edge switches still making sure to exclude all aggregation switches in the working and backup paths. We select the path with minimum additional power consumption.

by the end we should find a second backup path, if we cannot the algorithm will show an error message. if we do have a path we activate the elements in it if necessary.

BPath2 function:

```
function pc=Bpath2(co,s,j,cap,ag,ag2)
    p=list();
    pc = struct('capacity',-1,'path',p);
    pc.path=list();
    res=jinvoke(co,"getLabel");
    if isequal(res,s) then
        pc.path(j)=res;
        pc.capacity=cap;
    elseif ~isequal(res,ag) & ~isequal(res,ag2) & jinvoke(co,"hasChildren") then
        for z=0:(jinvoke(co,"getNChild")-1)
            nod=jinvoke(co,"getChild",z);
            bol=jinvoke(nod,"getAct");
            if bol then
                aux=Bpath2(nod,s,j+1,jinvoke(co,"getCapacity",z),ag,ag2);
                if ~isequal(aux.capacity,-1) then
                    pc.path(j)=jinvoke(co,"getLabel");
                    for m=j+1:size(aux.path)
                        pc.path(m)=aux.path(m);
                    end
                    pc.capacity=min(aux.capacity,cap);
                end
                jremove(res);
            end
        end
    end
endfunction
```

Bpath2 function is a recursive function in charge of finding a path connecting the core node co and the edge switch s that does not contain ag or ag2. The function ends when the base case is met, that is when the edge switch s has been reached while exploring the children of the core node co. The function returns the path found and the capacity it has available calculated as the minimum capacity of the links that form the path.

actBpath2 function:

```
function pc=actBPath2(co,s,j,cap,ag,ag2)
    pc = struct('power',-1,'capacity',-1,'path',list());
    pc.path=list();
    res=jinvoke(co,"getLabel");
    if isequal(res,s) then
        pc.path(j)=res;
        pc.capacity=cap;
        pc.power=jinvoke(co,"getP")+(jinvoke(co,"getNact")
        *jinvoke(co,"getPport"));
    elseif jinvoke(co,"hasChildren") & ~isequal(res,ag) &
        ~isequal(res,ag2) then
        for z=0:(jinvoke(co,"getNChild")-1)
            nod=jinvoke(co,"getChild",z);
            cap=jinvoke(co,"getCapacity",z);
            aux=actBPath2(nod,s,j+1,cap,ag, ag2);
            if ~isequal(aux.capacity,-1) then
                pc.path(j)=jinvoke(co,"getLabel");
                for m=j+1:size(aux.path)
                    pc.path(m)=aux.path(m);
                end
                if jinvoke(co,"getAct") then
                    pow=jinvoke(co,"getP")+((jinvoke(co,"getNact")+1)
                    *jinvoke(co,"getPport"));
                else
                    pow=jinvoke(co,"getP")+(jinvoke(co,"getNact")
                    *jinvoke(co,"getPport"));
                end
                pc.power=aux.power+pow;
                pc.capacity=min(aux.capacity,cap);
            end
            jremove(res);
        end
    end
endfunction
```

actBpath2 is the function in charge of browsing all paths in the network in case none of the active were suitable. It is a recursive function that finds the path with minimum additional power consumption that connects the core switch co and the edge switch s and does not include the aggregation switches ag or ag2. The function ends when the edge switch is reached. Otherwise we recursively explore all children of the core co as long as they aren't the aggregation nodes ag or ag2.

CODE FOR THE INPUT GENERATORS

INPUT GENERATOR FOR FAR TRAFFIC

```
k=int32(8);
c=10;
p=int32(146);
pport=0.9;
n=25;
t=3;
wri=string(k)+ascii(9)+string(c)+ascii(9)+string(p)+ascii(9)
+string(pport)+ascii(9)+string(n)+ascii(9)+string(t);
wrii=string(k)+ascii(9)+string(c)+ascii(9)+string(p)+ascii(9)
+string(pport)+ascii(9)+string(n)+ascii(9)+string(t);
fd=mopen('C:\Users\Elena\Desktop\ElasticTree\Input.txt','w');
fd2=mopen('C:\Users\Elena\Desktop\ElasticTree\InputF.txt','w');
npod=double(k);
nedg=double(k/2);
ns=nedg;
serv=double((k^3)/4);
for i=1:200
    pod=int32(rand(1)*npod+1);
    while(pod==0)
        pod=int32(rand(1)*npod+1);
    end
    pod2=int32(rand(1)*npod+1);
    while(pod2==0 | pod2==pod)
        pod2=int32(rand(1)*npod+1);
    end
    disp(pod, pod2);
        edg1=int32(rand(1)*nedg+1);
    edg2=int32(rand(1)*nedg+1);
    while(edg1==0)
        edg1=int32(rand(1)*nedg+1);
    end
    while(edg2==0)
        edg2=int32(rand(1)*nedg+1);
    end
    s1=int32(rand(1)*ns+1);
    while(s1==0)
        s1=int32(rand(1)*ns+1);
    end
```

```
    end
    s2=int32(rand(1)*ns+1);
    while(s2==0)
        s2=int32(rand(1)*ns+1);
    end
    s=int32(((pod-1)*nedg*ns)+((edg1-1)*ns)+s1);
    d=int32(((pod2-1)*nedg*ns)+((edg2-1)*ns)+s2);
    b=rand(1)*(c*0.1);
    b=round((b*100))/100;
    while(b==0)
        b=rand(1)*(c*0.1);
        b=round((b*100))/100;
    end
    wri=wri+ascii(9)+string(s)+ascii(9)+string(d)+ascii(9)+string(b);
    wrii=wrii+ascii(10)+string(s)+ascii(9)+string(d)+ascii(9)+string(b);
end
    mputl(wri,fd);
    mputl(wrii,fd2);
    mclose('C:\Users\Elena\Desktop\ElasticTree\Input.txt');
    mclose('C:\Users\Elena\Desktop\ElasticTree\InputF.txt');
```

INPUT GENERATOR FOR MIDDLE TRAFFIC

```
k=int32(8);
c=10;
p=int32(146);
pport=0.9;
n=10;
t=2;
wri=string(k)+ascii(9)+string(c)+ascii(9)+string(p)+ascii(9)
+string(pport)+ascii(9)+string(n)+ascii(9)+string(t);
wrii=string(k)+ascii(9)+string(c)+ascii(9)+string(p)+ascii(9)
+string(pport)+ascii(9)+string(n)+ascii(9)+string(t);
fd=mopen('C:\Users\Elena\Desktop\ElasticTree\Input.txt','w');
fd2=mopen('C:\Users\Elena\Desktop\ElasticTree\InputM.txt','w');
npod=double(k);
nedg=double(k/2);
ns=nedg;
serv=double((k^3)/4);
for i=1:200
    pod=int32(rand(1)*npod+1);
    while(pod==0)
        pod=int32(rand(1)*npod+1);
        end
    edg1=int32(rand(1)*nedg+1);
    edg2=int32(rand(1)*nedg+1);
    while(edg1==0)
        edg1=int32(rand(1)*nedg+1);
    end
    while(edg2==0 | edg2==edg1)
        edg2=int32(rand(1)*nedg+1);
    end
    s1=int32(rand(1)*ns+1);
    while(s1==0)
        s1=int32(rand(1)*ns+1);
    end
    s2=int32(rand(1)*ns+1);
    while(s2==0)
        s2=int32(rand(1)*ns+1);
    end
    s=int32(((pod-1)*nedg*ns)+((edg1-1)*ns)+s1);
    d=int32(((pod-1)*nedg*ns)+((edg2-1)*ns)+s2);
    b=rand(1)*(c*0.1);
    b=round((b*100))/100;
    while(b==0)
        b=rand(1)*(c*0.1);
        b=round((b*100))/100;
    end
    wri=wri+ascii(9)+string(s)+ascii(9)+string(d)+ascii(9)+string(b);
    wrii=wrii+ascii(10)+string(s)+ascii(9)+string(d)+ascii(9)+string(b);
end
    mputl(wri,fd);
    mputl(wrii,fd2);
    mclose('C:\Users\Elena\Desktop\ElasticTree\Input.txt');
    mclose('C:\Users\Elena\Desktop\ElasticTree\InputM.txt');
```

INPUT GENERATOR FOR MIXED TRAFFIC

```
k=int32(6);
c=10;
p=int32(146);
pport=0.9;
n=25;
t=1;
wri=string(k)+ascii(9)+string(c)+ascii(9)+string(p)+ascii(9)
+string(pport)+ascii(9)+string(n)+ascii(9)+string(t);
fd=mopen('C:\Users\Elena\Desktop\ElasticTree\Input.txt','w');
wrii=string(k)+ascii(9)+string(c)+ascii(9)+string(p)+ascii(9)
+string(pport)+ascii(9)+string(n)+ascii(9)+string(t);
fd2=mopen('C:\Users\Elena\Desktop\ElasticTree\InputMx.txt','w');
serv=double((k^3)/4);
for i=1:1
    s=int32(rand(1)*serv);
    while(s==0)
        s=int32(rand(1)*serv);
    end
    d=int32(rand(1)*serv);
    while(d==0 | d==s)
        d=int32(rand(1)*serv);
    end
    b=rand(1)*(c*0.1);
    b=round((b*100))/100;
    while(b==0)
        b=rand(1)*(c*0.1);
    b=round((b*100))/100;
    end
    //wr=[string(s),string(d),string(b)];
    //wri=cat(1,wri,wr);
    wri=wri+ascii(9)+string(s)+ascii(9)+string(d)+ascii(9)+string(b);
    wrii=wrii+ascii(10)+string(s)+ascii(9)+string(d)+ascii(9)+string(b);
end
    mputl(wri,fd);
    mputl(wrii,fd2);
    mclose('C:\Users\Elena\Desktop\ElasticTree\Input.txt');
    mclose('C:\Users\Elena\Desktop\ElasticTree\InputMx.txt');
```